クラスタコンピューティング向け 高速ネットワークカードの Linuxデバイスドライバの開発

A development of a Linux device driver of Fast Network Card for Cluster Computing

鶴崎 剛大

平成12年2月15日

我々は分散システム「コンピュータ・コロニー」を提案している。コンピュータ・コロニーは並列アプリケーションの実行を指向しており、従来の分散システムに欠けている高スループット、低レイテンシの通信機構、資源の位置透過性、公平な資源の分配を目標とするものである。

コンピュータ・コロニーは高速なネットワークカードで結合された複数のワークステーションからなるワークステーションクラスタと、オペレーティングシステムColoniaで構成される。要素となるワークステーションはそれ単体でも動作し、相互に接続されれば完全にネットワーク透過で並列アプリケーションを支援する環境となる。

現在、コンピュータ・コロニーのプロトタイプが開発されており、ハードウェアはSPARCstation20とシステムバスに接続されるFibre Channelカードとからなっている。ソフトウェアはUNIX上で動作する“エミュレータ”を使用したColoniaの上位部分と、ハードウェアに密着した下位部分とからなっている。

このプロトタイプの通信機構は共有メモリをベースとした通信プリミティブを提供しており、ユーザーに安全に通信を解放することができる。現在、その通信機構の基本部分について実装が進められているが、将来におけるデータ通信のより一層の高速化を目標とした、通信方式の検討も行なわれている。その中の1つに、ユーザープログラムからのブロック転送の利用があり、DMAによってCPUを消費せず高速にデータ転送を行なう機能を、システムコールを介さないで利用させることを考えている。本研究では、そのDMA転送機能のみを持つコンピュータ・コロニー通信ボードのサブセットを想定し、その機能のユーザーアプリケーションへの解放を模索した。

また、プロトタイプのソフトウェアでは開発コストの削減のためにSPARC/Linuxのデバイスドライバをそのまま利用することを考えている。このためにLinuxのデバイスドライバモデルについての調査も進められている。

本研究では、Linuxのデバイスドライバモデルについての理解を深めるために、前述のネットワークカードのLinuxデバイスドライバを開発した。

本稿ではまず、最大1Mbytesのブロックを DMA(Direct Memory Access)転送可能なネットワークカードの仕様を定める。そしてそのネットワークカードを用いてメモリ共有を実現する手順を示す。最後に、Linuxが稼働する異なるノード上のプロセス間でのメモリ共有を利用するための、Linuxデバイスドライバを作成し、その詳細について説明する。

このデバイスドライバは、ネットワークカードのLinuxにおける名前となる*/dev/nicdma*、ユーザー空間にDMAバッファをマッピングするための*open, mmap, close*システムコール、およびDMA転送の終了をユーザープロセスに知らせるための割り込みハンドラ、を提供する。ユーザープロセ

スは、*/dev/nicdma* を *open* した後、*mmap* し、別ノードのプロセスとメモリの共有を実現する。そして必要に応じて *mmap* された領域への書き込みを行ない、DMA および通信を起動する。通信の起動は、メモリ領域にマッピングされたネットワークカードレジスタへの書き込みによってなされ、システムを介する必要はない。また通信中もプロセスは動作を続行することができるため、通信に関するソフトウェアオーバヘッドはほぼ存在しない。DMA 転送が終了すると、ネットワークカードはDMA 中であることを示すフラグをクリアする。また、設定によっては割り込みをかけることもできる。割り込みハンドラはこの割り込みによってプロセスにシグナルを送るため、通信の終了は、レジスタの値をチェックするか、シグナルの到着によって知ることができる。

We are proposing distributed system "Computer Colony". The Computer Colony is a system for efficient execution of parallel application, and its goal is high throughput and low latency communication mechanism, location-transparency, and fair sharing of resources, which a conventional distributed system lacks.

The Computer Colony consists of a workstation cluster which is connected with a high-speed network and the operationg system Colonia. An element workstation works by its own and becomes a environment to support parallel application totally and network-transparent execution when they linked mutually.

At present, a prototype of the Computer Colony is under development and its hardware consists of SPARCstation20 and Fibre Channel card which is connected with the system bus. Its software consists of a upper layer of the Colonia developed with "emulator" which works on UNIX and a lower layer stucks to the hardware.

Communication mechanism of this prototype is providing communication primitives based on shared memory and can offer communication safely to the user. At the present time, we develop basic parts of the communication mechanism. And we examine communication methods which makes data communication faster in the future, too. One of the communication methods is utilization of function of block transfer, that a user program transfers data on DMA at high speed without consuming CPU. And we want to make available of the communication method without system calls. In this study, I fixed specification of a subset which has only tranfering function with DMA of that the Computer Colony communication board does. And I groped for making available the function to user application.

Also, we think of using the device driver of SPARC/Linux with prototype software for a reduction of the development cost. So investigation about the device driver model of Linux, too, is carried forward for this purpose.

In this study, to deepen understanding about the device driver model of Linux, I developed a device driver of an above-mentioned network card.

First, this thesis fixes a specification of the network card that can transfer with DMA(Direct Memory Access) a memory block within 1Mbytes. Then, the procedure which realizes memory sharing using the network card is shown. Finally, I make a Linux device driver which is to use memory sharing between processes, and give a detail of the device driver.

This device driver provides */dev/nicdma* which is a name in Linux, *open, mmap, close* system calls which is utilized to map DMA buffer in user space, and a interrupt handler to inform user process that a transfer ends. User process *open*es */dev/nicdma*, *mmap*s that, and then makes memory sharing with a process on other node. Then, it writes in a memory area which was *mmap*ed and it starts DMA and communication. Communication starts by writing in network card registers which was mapped in a memory area and there is no necessity of a service of a system. And, during communication, too, it is possible to continue an operation by the process, so software overheads about the communication doesn't exist approximately. When the DMA transfer ends, the network card clears a flag bit which shows to

be in DMA. Also, depending on setting, it can interrupt too. An interrupt handler sends a signal to the process by this interrupt. So, the process can become aware of an end of a transfer by checking a flag or by receiving a signal.

目次

1 はじめに

現在、研究室やオフィスなど、ある程度の計算機資源がある環境には分散システムが広く普及している。これらの分散システムでは日常的な軽い仕事は問題なくこなせるが、基本的にネットワーク越しに並列処理をこなせるようには作られていない。最近では 1Gbps もの高バンド幅を持つ汎用ネットワークも利用可能になっているが、これは汎用データ通信のためのネットワークであり、高スループットではあってもレイテンシは大きい。このため、粒度の粗い並列処理ならばこなせるものの、やはり一般の並列アプリケーションを実行させるには問題がある。また、プログラミングインターフェースも並列アプリケーション向きではなく、ネットワーク越しに並列アプリケーションを実行させようとするとプログラマに多大な負担がかかることにもなる。

分散システムにおいて、より細粒度の並列プログラムをネットワーク越しに動かすためにはソフトウェア環境及びハードウェア環境の両方で改善が必要である。

我々の提案している分散システム「コンピュータ・コロニー」は、高速、低遅延な共有メモリを安全にユーザーに解放するためのハードウェアと、並列アプリケーションの実行を強力に支援するオペレーティングシステム (Operating System, OS) とからなる総合的な環境である。コンピュータ・コロニーはマルチユーザーで利用され、ユーザーは他のユーザーに気がねすることなく並列アプリケーションを実行することができる。

このコンピュータ・コロニーの開発はハードウェア、ソフトウェアの両面から進められており、その中でハードウェアと密接に関わる部分の開発では、開発コストを軽減するために既存の OS(SPARC/Linux[1]) のデバイスドライバを流用することを考えている。SPARC/Linux のデバイスドライバを流用する仕組みが完成すれば Linux のデバイスドライバはそのままコンピュータ・コロニーで利用できるようになる。

また、コンピュータ・コロニーの通信ハードウェアではその通信方式として、DMA(Direct Memory

---

[1] SPARC アーキテクチャの WS で動作する Linux
`http://www.ultralinux.org/`

Access) を利用したブロック転送の機能をユーザーに解放することを考えている。

本研究では Linux デバイスドライバの構造を知る目的も兼ねて、その通信ハードウェアのブロック転送をユーザーに解放するための Linux デバイスドライバの開発を行なう。

本論文では第 2 章でコンピュータ・コロニーについて概説し、第 3 章でコンピュータ・コロニーにおける高速通信機構、および DMA によってブロック転送を実現する機構について説明する。そして第 4 章で実装した Linux デバイスドライバの詳細を説明する。

2　コンピュータ・コロニー

本章では分散システム「コンピュータ・コロニー」の概要を述べる。

「コンピュータ・コロニー」とは「群体 (コロニー)」のごとく、単体でも動作するコンピュータが多数結合してあたかも 1 台の並列計算機のように機能する、この分散システムを指す名称である。

2.1　概要

分散システム「コンピュータ・コロニー」の目指すものは、利用者にとって快適な分散コンピューティング環境である。この研究の背景とその目標を以下で説明する。

2.1.1　背景

現在、計算機の演算能力は急速に高まり続けており、1CPU のパーソナルコンピュータでさえ、個人の日常の仕事をこなすのに十分な能力を持つに至っている。しかしながら、こういった性能の向上に呼応するように、より高い性能を必要とする応用が生まれつつあり、その要求演算能力の上昇はとどまるところを知らない。このため将来はより高い演算能力をより手軽に、そして安全に引き出せる計算機環境が必要になるだろう。

こうした要求を満たす解として並列計算機が登場してからずいぶん経ち、並列計算機は集中システムと分散システムに分化してきた。

集中システムでは高速な通信機構が利用できるが高価であり、またシステムの拡張性に難があるため導入できるワークグループは限られており、分散システムは価格も手頃であるため導入が容易であるが、集中システムほどの通信性能は望むべくもない。

しかし、現在の分散システムは効率的な並列アプリケーションの実行を指向していないために通信のオーバヘッドが大きい、負荷分散がうまく行なえない等問題がある。こういった点を改善することで、集中システムに匹敵する実性能を発揮する分散システムを構築することが可能であると我々は考えている。

2.1.2　目標

コンピュータ・コロニーの目標は現在の分散システムの利点を保ったまま、より高い並列性能を手に入れることである。現在の分散システムの利点とはすなわち、

1. **導入の容易さ** 単体でも動作し、また構成が自由であるために初期導入コストを低く抑えることができる。
2. **利用の気軽さ** 基本的に 1 台の計算機は個人の裁量で使うことができ、重たいジョブを投入しても他人に気がねしなくてよい。
3. **耐故障性** 故障した際には故障した要素計算機のみを切離して交換、あるいは修理をすればよく、切離すことで最大性能は下がるものの機能は失われないため高い耐故障性を持つ。

という点である。
コンピュータ・コロニーでは、1についてはさらに

- マシン追加によってプロセッサ台数を増やすことができ並列性能の向上が見込める。

と期待することができる。
2については、

- ひとりに与えられる資源の量を制限し、公平な計算機資源の配分を行なうことで、重い並列ジョブを投入しても他のユーザーが不便を強いられないようにする。

ということを目標に加える。

これらの利点を保ったままでより高い並列性能、すなわち同等の理論最大性能を持つ単一の並列機に匹敵する並列処理能力、を獲得することが目標である。

2.1.3 要求仕様

目標を達成するためには以下の要求を満たさなくてはならない。

高速・低遅延な通信機構

比較的細粒度の並列処理で実効性能を出すには、可能なかぎり通信のオーバヘッドを減少させる必要がある。また、マルチユーザー環境であるので、通信は安全に行なわれなければならない。

現在の分散システムで通常用いられている通信機構は、単純なI/Oを行なうネットワークカードと、通信資源の保護を行なうシステムプログラム、そしてTCP/IPなどの汎用プロトコルからできており、システムの介在およびプロトコル処理のオーバヘッドは、CPUのサイクル数で数千から数万サイクルにもなる。

コンピュータ・コロニーがその目標を達成するためには、こういったソフトウェアのオーバヘッドを削減する必要がある。

資源の位置透過性

システム全体の利用効率を上げるためには、プログラムが位置によらず実行できる必要がある。計算機資源の名前がそれを見る位置によって異なっていれば、プログラムは一定の位置でしか実行できなくなるためである。また、プロセスを動的に移送できればさらに利用効率を上げられる。

2.2 コンピュータ・コロニーの開発状況

コンピュータ・コロニーは、上記の目標を達成するためにプロトタイプを開発中である。ここで、その開発状況について述べておく。

現在、コンピュータ・コロニーの開発は主にハードウェアとソフトウェアとに分けて進められている。ソフトウェアの開発はまた、OSとしてのサービスを提供する部分とハードウェアに密着した部分とに分かれて進められている。この理由は次のとおりである。

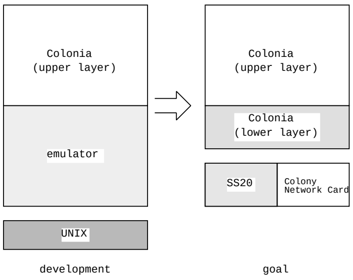


図 1: エミュレータによる開発

本来、OSはハードウェアや、ハードウェアに密着した下位の部分がなければ動作できないし、ハードウェアに密着した部分と切離してOSの動作を規定することもできない。しかしそれらの部分が完成する前にオペレーティングシステムColoniaとしての特徴的な部分である公平なスケジューリングや、資源の位置透過性、といった部分の開発を行なうため、UNIX上でこれらの代わりをする“エミュレータ”を作成して上位部分の開発を進めている。(図1)

ハードウェアに密着した下位の部分については、開発コストの軽減のためSPARC/Linux(以下単にLinuxと記す)の豊富なデバイスドライバの利用を考えている。

ここで、

- Linuxのデバイスドライバについて知り、
- コンピュータ・コロニーのネットワークカードとOSとはいかにして通信すればよいのかを知り、
- またColoniaのデバイスドライバモデルがどうあるべきかを知る

ために、まずハードウェアで共有メモリを管理するネットワークカードのためのLinuxデバイスドライバを作成することにした。

3 高速通信機構

コンピュータ・コロニーでは現在プロトタイプの開発が行なわれている。このプロトタイプでは

システムバスに接続する専用通信ボードを用いて、ハードウェアによる共有メモリを実現する。この通信ボードは通信機構をユーザーに解放し、ソフトウェアオーバヘッドの極力少ない通信を提供する。

また、DMA 機構をユーザーに解放することで、必要なデータを前もって他のノードに転送することができ、かつ転送中に、CPU は他の仕事をすることができる。このため、並列ソフトウェアのより一層の高速化が図れるようになっている。

本研究ではこの専用通信ボードと同様に、DMA 機構をユーザーに解放しCPU を消費せずにブロック転送を行なえるネットワークカードを想定し、これを利用するためのデバイスドライバを Linux 上で開発する。

3.1でコンピュータ ・コロニーの高速通信機構の概略を述べた後、3.2において本研究の Linux デバイスドライバが想定するネットワークカード (コンピュータ ・コロニーの専用通信ボードのサブセット) の仕様を説明する。

3.1 コンピュータ ・コロニーの高速通信機構

2で述べた目標を達成するために、コンピュータ ・コロニーのプロトタイプではその通信機構に共有メモリベースの通信を選択した。また、通信ボードはI/O バスではなくシステムバスに接続することで、共有メモリへのアクセスをハードウェアが自動的に処理できるようになり、通信遅延の低減が可能になった。

3.1.1 コンピュータ ・コロニーにおけるメモリ共有

コンピュータ ・コロニーでは、ノード間にまたがった並列プログラムの実行に適したモデルとして、ミッション ・ユニットモデルが提案されている。

ユーザーの代理として並列プログラムを実行する実体が 「ミッション」であり、これを細分化した並列処理単位が 「ユニット」である。ユニットは異なるマシンに分かれて同時に稼働することができる。

同一ミッションに属するユニットは同一アドレスで共有空間を持っており、共有データやプログラムコードはすべて同一アドレスになっている。このため、ポインタをそのまま他のユニットに渡して利用することができる。異なるミッション間でメモリを共有する場合は特にアドレスが同一になるということはない。

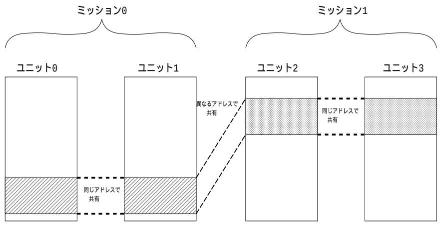


図 2: ミッション ・ユニットモデル

コンピュータ ・コロニーでは専用通信ボードがアドレス変換を行なうため、共有空間の仮想アドレスが同じであれば実アドレスが異なっていても自動的に異なるノード間で共有空間のコンシステンシが維持される。

3.1.2 通信ボードの構成

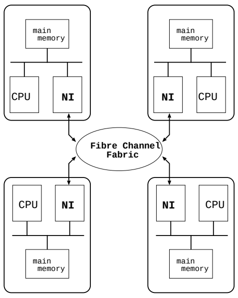


図 3: プロトタイプのネットワーク構成

図 3にプロトタイプのネットワーク構成を示す。SPARCstation20(SS20) に専用通信ボードを搭載したマシンがプロトタイプハードウェアの 1 ノードとなる。各ノードではオペレーティングシステム Colonia が動作しており、ノード間共有メモリは通信ボードの支援を得て、Colonia が実現する。共有メモリの参照は比較的高速ではあるが、ローカルノードの主記憶を参照するほうが数段高速である。このため、リモートノード上に実体がある共有メモリを一部ローカルノードにキャッシングしてさらなる高速化を図っている。

各ノードの通信ボードの構成は図4のようになっている。

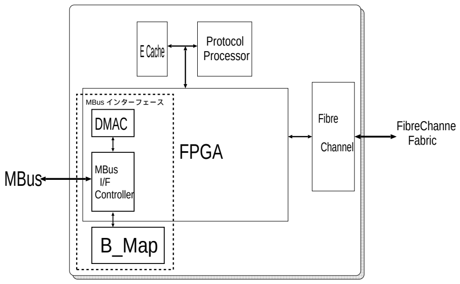


図 4: Computer Colony 向け通信ボードの構成

この専用通信ボードの機能の大半は FPGA によって実現される。また、プロトコルプロセッサを実装しており、柔軟かつ強力に様々な機能を実現できる。

MBus インターフェース部にある B_Map は、主記憶のタグになっている。このタグには、ローカルノードに実装されている物理メモリのブロック[2]数ぶんだけエントリがあり、そのエントリに、共有、無効の各状態を保持している。

この通信ボードは MBus トランザクションをスヌープしており、MBus にトランザクションが発生するとそのアドレスで B_Map を引き、その結果によって通信の必要性を判断する。通信の必要性を認めた場合は自動的に通信を起動し、リモートノードの共有関係にあるメモリブロックを転送し、MBus トランザクションのアドレスに対するデータとして MBus に流す。

この一貫性制御は32bytes と比較的小さなブロック単位で行なわれるため、ある程度大きなサイズでメモリの書換えが発生するような状況では効率の低下が懸念される。つまり、リモートノードで読み込みが発生するたびにブロック単位で通信するよりも、データの生産者側で(32bytes ブロックよりも大きな単位で) 明示的に転送を指示してやった方が高速であろう、ということである。

このような状況を想定して、この通信ボードでは DMA 転送機能を装備している。ユーザーが DMA を起動すると、通信ボードが自動的に主記憶を読み出し、リモートノードの対応する共有空間へと通信を行なう。この機構はユーザーに解放されており、システムの介在なしに実行できる。

以下ではこの DMA 転送機能に着目し、通信ボードのサブセットとしてユーザーが明示的に DMA 転送を起動できるような機能を持つネットワークカードを想定する。そしてそのネットワークカードをユーザーが利用できるようにするための、OS - ユーザー間のインターフェースとして、デバイスドライバを Linux 上で実装する。

3.2 Linux で実現する通信機構

以降の章では、Linux 上でノード間共有メモリを実現することを考え、そのためのネットワークカードの仕様を定める。このネットワークカードの仕様はコンピュータ ·コロニーの通信ボードの仕様のサブセットであるため、ブロック単位でのコンシステンシの維持やアドレス変換の機能は持たないものとする。ただし、ユーザーが DMA を起動し、自動的に連続した領域を別のノードに転送する機能は持っているものとする。つまり、ここで実現されるメモリ共有はユーザーからの明示的な指示によって同期される、共有データ領域である。

3.2.1 DMA を介したメモリ共有

このサブセットカードは、物理メモリの一部を DMA バッファとして固定的に割り当てることを想定している。つまり、一度起動すると終了するまでその領域は解放されず、そのアドレスも変わることがない。

この DMA バッファを、ユーザープログラムが *mmap* することによってプログラム間のデータ共有が実現される。(図 5)

また、DMA をコントロールするためのレジスタはユーザー空間に *mmap* され、ユーザーがアドレスとサイズを指定して DMA start bit をセットすることで通信が起動する。ここで用いるアドレスは DMA バッファ内オフセットであるが、プロセス内の仮想アドレス、DMA バッファとして確保された実メモリ共にリニアアドレスであるため、仮想アドレス同士の減算によって容易に求めることができる。

---

[2] 他のノードと共有しているか否かを通信ボードが判断する、基本サイズ。この実装では 32bytes である。

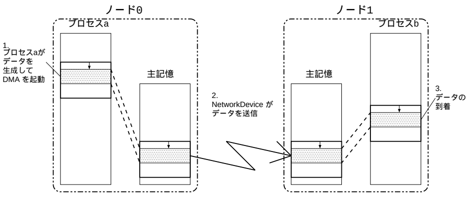


図 5: DMA を介したメモリ共有

次項でサブセットカードの仕様を述べる。

3.2.2 サブセットネットワークカードの仕様

本ネットワークカードは MBus モジュール[3]として SS20 に搭載される。開始アドレスとサイズをこのネットワークカードに通知すると DMA 転送が開始され、そのデータ領域がリモートノードに送信される。

このネットワークカードの概要を図 6に示す。

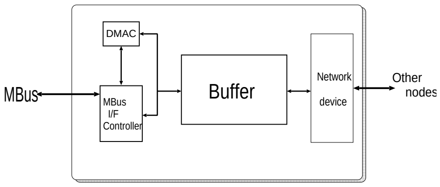


図 6: 想定するネットワークカードの構成

DMA バッファのサイズは 1Mbytes とし、Linux 起動時に固定的に割り付けられるものとする。デバイスドライバとネットワークカードはそのアドレスを知る必要があるが、ユーザープログラムは知る必要がない。また、アドレス指定はバッファ内のオフセットで指定されるため、他のノードの DMA バッファアドレスを知る必要もない。

MBus モジュールには 1 つのモジュールにつき 0xFFn000000 ~ 0xFFnFFFFFF, (n はモジュール ID, $0 < n \le 16$) までの 16Mbytes のアドレス空間が与えられる。その空間内のアドレス配置は次のとおり。

ネットワークカードの MBus Configuration

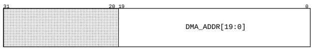
Address Map

MBus アドレス 0xFFn000000 ~ 0xFFnFFFFFF(n は MBus ID) は MBus デバイス用に予約されており、これら 16 個の空間に順次アクセスすることで、システムは MBus デバイスの存在とその種類を知ることができる。また、システムと MBus デバイス間のデータ転送にも用いられる。

1. DMA コントロールレジスタ 0

   0xFFn000000 からの 4bytes

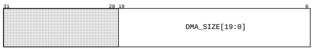
   DMA コントロールレジスタ 0 および 1 はユーザー空間にマッピングされてユーザープログラムにより制御される。

   - DMA_ADDR[19:0]
     DMA を開始するアドレス 20bits.
     DMA を開始するアドレスを DMA バッファ先頭からのオフセットで指定する。

2. DMA コントロールレジスタ 1

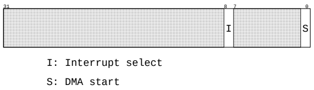
   0xFFn000004 からの 4bytes

   - DMA_SIZE[19:0]
     DMA 転送を行なうサイズ。

3. DMA コントロールレジスタ 2

   0xFFn000008 からの 4bytes

   - interrupt select
     DMA 完了時にカードが割り込みをかけるかどうかを指定する。
     0:割り込みをかけない (デフォルト)
     1:割り込みをかける

---

[3] DMA が利用できることがこのネットワークカードの要件であるので、SBus モジュールとしてもかまわない。ここではデバイスドライバ実装の都合上、MBus モジュールとした。

- DMA start
  DMA start に値 1 を書き込むと直ちに DMA を開始する。DMA が終了するとネットワークカードによってクリアされる。

4. 送信バッファ

   0xFFnFFFFF0 からの 4bytes

   このバッファは DMA バッファの開始アドレスなどを OS がカードに送信するために用いる。

   これ以降の送信バッファ、受信バッファ、NSR, MPR はデバイスドライバによって用いられ、ユーザープログラムがアクセスすることはない。

5. 受信バッファ

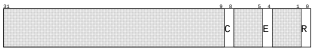
   0xFFnFFFFF4 からの 4bytes

   このバッファはカードから OS への通信に用いられる。

6. Networkcard Status Register(NSR)

   0xFFnFFFFF8 からの 4bytes

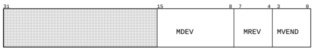
   ネットワークカードの状態を示し、状態を制御するためのレジスタである。このレジスタのフィールドの意味は以下のとおり。



   - C(action)
     Reset and Clear (write only).
     このビットをセットすると、ネットワークカードはリセットされて初期状態に戻る。
   - E(state)
     Error.
     通信デバイスの回復不能な error によってセットされる。
   - R(state)
     Ready.
     電源投入後、ネットワークカードの初期化が終了してシステムからのデータを受けつけられるようになるとセットされる。システムはこのビットがセットされるのを待って作業を開始する。

7. MBus Port Register(MPR)

   MBus デバイスの存在とその種類を示すレジスタ。この MPR の位置は 0xFFnFFFFFC からの 4bytes に定められている。ここにはネットワークカード固有の ID を格納しておかなければならない。



   - MDEV
     MBus デバイス番号。本ネットワークカードのデバイス番号を格納する。このカードでは 0x00 を格納するものとする。
   - MREV
     デバイスリビジョン番号。ネットワークカードのリビジョンアップに伴なって値は変化させるべき。当実装では 0x0。
   - MVEND
     MBus ベンダ番号。デバイスベンダの番号を指定する。すでに存在するベンダ番号と重複すると問題が発生するので、当実装では,0x7 を選択した。

3.2.3 ユーザープロセスによる DMA の起動

この DMA 転送は以下の状況下で行なわれることを想定している。

- 異なるノード間に、プロセス a および b があり、a はデータの生産者で (メモリに書き込みを行なう)、b はデータの消費者である (メモリへの書き込みは行なわない)。

- プロセス a と b とは共に DMA バッファ領域を *mmap* している。同時に DMA コントロールレジスタも *mmap* している。

Linux 上であるから、プロセス a と b とは同一のプログラムでなくてもよい (例えば、a は鍵ブロック生成プログラムで b はその鍵ブロックをチェックする暗号解読プログラム etc.)。

2 つ以上のプロセスが DMA バッファを *mmap* すると、それらのプロセス間ではリモートノード間のデータ共有が行なえるようになる。

このネットワークカードでブロック転送を行なう手順は次のようになる。

- ユーザープロセスが DMA コントロールレジスタ 0,1 および DMA バッファを mmap する。
- DMA を開始するアドレスを DMA コントロールレジスタ 0 にセットし、DMA サイズ & DMA start bit をセットすると、直ちに DMA が開始される。ユーザープロセスの実行はそのまま続行される。
- ネットワークカードは指定された領域を読み出し、リモートノードに送信する。リモートノードのカードは、MBus を占有して自分が確保している DMA バッファの先頭アドレス+オフセットの領域の実メモリに書き込みを行なう。

以上で転送が行なえる。

3.2.4 DMA 転送の終了

DMA 中は、MBus の帯域幅の一部が占有されるものの、CPU は動作可能である。このためユーザープロセスは動作を続けることができると期待される。これはパフォーマンスを改善するものの、DMA 中の領域にもアクセスできてしまうため、リモートノードに間違ったデータを送信してしまう危険を孕んでいる。危険回避のためにはユーザープログラムが DMA の開始と終了を正しく把握する必要がある。

このネットワークカードのデフォルトの動作では、DMA の終了を知るためにユーザープログラムが DMA start bit を調べるしかない。この場合、DMA バッファを A, B 2 つに分けて

1. 値を計算して A に格納する。
2. DMA 中かどうかチェックして DMA 中であれば 2. へ、そうでなければ 3へ。
3. 開始アドレスとサイズを A を指すようにセットして DMA 転送の開始を指示
4. 計算を B 上で続行する。
5. 終了したかどうかチェックして、終了していれば 6. へ、そうでなければ少し待って 5. へ。
6. 開始アドレスとサイズを B を指すようにセットして DMA 転送の開始を指示。1へ。

というように若干の工夫をする必要がある。

実際には、ある領域にプロセスが値を計算して格納するよりも DMAC がその領域をアクセスし終わるまでの時間の方が短かいと思われるので、問題になる可能性は低いと思われる。

しかし、このネットワークカードでは DMA バッファ全体にかかるような巨大な領域を転送することも考えて、割り込み生成ができることを想定する。

このネットワークカードが装着される MBus はプロセッサとメモリとを接続することを主に考えている。このため、プロセッサに割り込みをかけることはできても、MBus モジュールから割り込み要求を発生させることができない可能性を考慮して、MBus アーキテクチャを確認した。その結果、MBus には I/O モジュールが実装される可能性も考慮されており、また、CPU から CPU へ割り込みを転送できるように、MBus モジュールの割り込み要求がサポートされていることを確認した。(図 7)

割り込みは、I/O モジュールのための割り込み要求線 INTOUT を使うか、SEC(SBus-to-EBus Controller) 内に実装されているシステム割り込みコントローラのレジスタにアクセスすることでかけることができる。

割り込みを利用する場合は、カーネルにインストールした割り込みハンドラがプロセスにシグナルを送信してくるのを待つだけである。ユーザープロセスはシグナルハンドラを実装することで、より効率よく通信が行なえると期待される。

次章ではこのネットワークカードをユーザープログラムから利用するための Linux デバイスドライバを作成する。

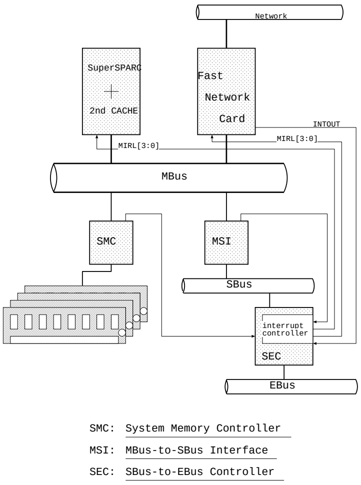


図 7: SS20 割り込みアーキテクチャ

4 Linux デバイスドライバ

本章では 3.2で定義したネットワークカードを Linux 上のユーザープロセスから利用可能にするためのデバイスドライバを作成する。

通常、ソフトウェアからハードウェアに対して入出力を行なおうとする場合、CPU で特権命令として扱われる操作が必要になる。現在の OS はユーザーアプリケーションを安全に実行するため(ユーザーアプリケーションのプログラムミスやエラーによってシステム全体が影響を受けることのないように) ユーザーアプリケーションには特権命令と呼ばれるいくつかの操作を許していない。このため特権の必要な I/O 入出力命令などを OS の中で実行するようにして、対象ハードウェアを抽象化したインターフェースをユーザーアプリケーションに解放するようにしている。このインターフェースを提供するための OS のサービスをデバイスドライバと呼ぶ。当然、対象とするハードウェアが違えばそれにアクセスするためのアルゴリズムも変わるため、デバイスドライバは存在する周辺機器の数だけ存在する。ただし、似た目的に使われるデバイスを利用するためのインターフェースは、なるべく同じであるのが望ましく、多くの OS で工夫がなされている。

ところで、特権状態と非特権状態との区別を持たない CPU アーキテクチャ上の OS や OS とユーザーとを区別しない OS であっても、入出力の抽象化によってインターフェースを統一するために、デバイスドライバは用いられる。また、入出力ではないカーネル内の雑務を行なうプログラムも近年ではデバイスドライバと呼ばれることもあり、デバイスドライバとは OS の中にあって、標準では提供されない機能を実現するルーチン全般を指す言葉に成りつつある。あるいは、ユーザープログラムであっても何らかの入出力操作を行なうプログラムをデバイスドライバと呼ぶこともあり、デバイスドライバと一口に言ってもそのイメージは 1 つには定まらなくなってきている。

しかしながら、ここでは

- 対象とする入出力装置を特定するための何らかの統一インターフェースによって、システム内で一意に定まる名前 (ex. /dev/*) を持ち、
- システムプログラムの一部として動作するもの

のみをデバイスドライバと称することにする。それ以外のユーザープログラムとして動作するものはユーザーモードドライバ、名前を持たないものは単にカーネルルーチンなどと呼んで区別することとする。

さて、ここでデバイスドライバの役割とは、

- 同種の入出力操作のインターフェースを統一して、プログラムからは同じように扱えるようにすること。
- ユーザーアプリケーションから危険な操作を分離して、システムプログラムの安全を守ること。

である。

UNIX ではユーザープログラムは、普段はハードウェアに対する直接アクセスやメモリに対する不正なアクセスが禁止されている、CPU の動作モード (ユーザーモード) で動作する。そしてシステムコールと呼ばれる有限数の操作によってのみ全ての操作が行なえる CPU の動作モード (スーパバイザモード、特権モード) に移行して入出力を行なうことができる。システムコールの前後ではきびしく値のチェックが行なわれ、またシステムコールのルーチンは吟味された OS の一部である

ため、OS はユーザープログラムから守られることができる。

今回作成したデバイスドライバは、*mmap* によって他のノードと共有可能なメモリ領域をユーザー空間にマッピングし、DMA によって CPU のあずかり知らぬ所で主記憶を書換える、という操作を安全にユーザーに解放するものである。

将来、コンピュータ・コロニーの開発が進めば、このデバイスドライバが行なう操作は、ハードウェアによってコンシステンシ維持がなされる共有メモリにおける機能、の一部として Colonia に取り込まれる予定である。

4.1 ネットワークドライバの構成

今回作成したドライバは、

- 初期化ルーチン

  mbus_nic_setup()

- *open, mmap, close* の各システムコールルーチン

  mshm_open()

  mshm_mmap()

  mshm_release()

- 割り込みハンドラ

  mbus_nic_interrupt()

から構成されている。

4.1.1 デバイスの名前づけ

デバイスに対して何らかの操作を行なうには、一意な名前が必要である。Linux において、*open, read, write, mmap,...* 等のシステムコールによって扱われるデバイスは、*/dev/*にファイルの形でその名前を保持している。これによって、デバイスは通常のファイルと同様のインターフェースでアクセスすることができる。このデバイスファイルが通常のファイルと異なる点は、これが 2 つの番号に関連づけられた、実体のないファイルである、という点である。これら 2 つの番号はメジャー番号、マイナー番号と呼ばれ、メジャー番号がデバイスの種類を、マイナー番号がデバイスの個体を表わしている。デバイスファイルがアクセスされた場合、メジャー番号によって対応するデバイスドライバのルーチンが呼び出され、マイナー番号はそのままデバイスドライバに渡されて個体を識別するために用いられる。つまり、ファイル名は本質的ではなく、番号がまったく同じで異なるファイル名のデバイスファイルは、まったく同じデバイスを指すことになる。

例えば、

```
crw-rw-rw-      1 root   root     6,       0 Nov 30
crw-rw-rw-      1 root   root     6,       1 Sep 18
crw-rw-rw-      1 root   root     6,       0 May  1
crw-rw-rw-      1 root   root     6,       0 Nov 30
```

のようにデバイスファイルが存在する場合、*/dev/lp0* と*/dev/lp1* とはメジャー番号が同じであるから、同じデバイスドライバによって扱われるが、マイナー番号が異なるため、異なる実体(パラレルポート) を指す。これに対して、*/dev/lp0, /dev/lpt0, /dev/printer* は全て番号が同一であり、どれにアクセスしても結果は全く同じになる。

新たなデバイスを追加する場合、すでに使われている番号と重複しないように、新規デバイス用番号を割り当てる必要がある。本稿のネットワークカードは、実験的用途向けに予約してある番号 120 ~ 127, 240 ~ 254 のうちから、254 をメジャー番号として用いる。個体の区別の必要はないため、マイナー番号は 0 を用いる。また、デバイスファイル名は*/dev/nicdma* とする。

このネットワークカードを利用するためには、

```
# mknod c /dev/nicdma 254 0
```

のようにしてデバイスファイルを作成しておかなくてはならない。

4.1.2 資源の確保

初期化ルーチンの最も重要な仕事はネットワークカードが要求する資源の割り付けである。

DMA バッファ

このネットワークカードは DMA バッファとして 1Mbytes の領域を要求する。この DMA バッファは物理アドレスでアクセスされるため、物理メモリから連続した 1Mbytes の領域を確保しなくてはならない。Linux では動作中の

カーネルに要求できる連続領域は 128Kbytes が最大となるため、カーネル起動時に認識された物理メモリを先頭から 1Mbytes 切取ることで解決している。この方式の欠点は、実際には DMA バッファが使用されていない時でもメモリを占有し続け、必要なくなった場合でも動作中の kernel にメモリを返すことができない点である。イメージとしては、あたかもマシンに実装されたメモリが 1Mbytes 減ったかのごとく見えることになる。ちなみに、この欠点はこのネットワークカードと Linux の実装のみに起因している。コンピュータ・コロニーでは、通信ボードはアドレス変換によって仮想アドレスを扱えるため、DMA バッファとして使用する領域は必ずしも物理的に連続していなくてもよい。

割り込み

ネットワークカードが要求するもう 1 つの資源は割り込みである。SS20 のシステムバス、MBus では、割り込みをかける方法が 2 通りあり、MBus 上の I/O デバイスが普通使用する方法 - MBus の信号線 INTOUT によって割り込み要求をする - では割り込み番号はデフォルトで 9 になる。[4]この割り込み線は SBus デバイスなどと共有されるため、共有ハンドラをインストールしなくてはならない。Linux では、共有された割り込みのハンドラをインストールするには

```
request_irq(irq, (*handler),
   SA_SHIRQ, *device, *dev_id);
```

のよう、割り込みハンドラをインストールする際に SA_SHIRQ を指定する必要がある。

ネットワークカードのデフォルトでは割り込みはかけないようになっているが、ユーザーアプリケーションによって割り込みをかけるよう選択することもできる。ユーザーが割り込みをかけることを選択した時点で割り込みハンドラをインストールすることはできないため、あらかじめインストールしておく必要がある。

4.1.3 ユーザー空間へのマッピング

ユーザーアプリケーションは仮想アドレスを用いるため、DMA バッファをマッピングする *mmap* システムコールを実装するには、ユーザープロセスの仮想→実アドレスを変換する部分の管理を行なわなくてはならない。

最終的に仮想→実アドレス変換を行なうのは MMU であり、アドレス変換情報を記載したページテーブルである。ページテーブルはまさにページ単位での変換テーブルであって、その操作はいくぶん繁雑になる。このため、Linux では VMA(Virtual Memory Area) と呼ばれる、より上位のメカニズムが実装されている。これは、同じ属性を持った連続アドレス領域ごとにまとめて取り扱うものである。デバイスドライバはこの VMA を通じて仮想アドレス領域を管理すればよい。*mmap* システムコールの実装においては、1 回の *mmap* につき VMA 構造体 1 つを保守するだけでよい。

このネットワークカードドライバの実装では、以下のものについてのみ保守を行なう。

unsigned long vm_start

unsigned long vm_end

1 つの仮想メモリ領域の開始アドレスと終了アドレス。この仮想アドレス範囲に DMA バッファがマップされるようにする。

*struct inode *vm_inode*

領域が i ノードに対応する場合、その i ノードへのポインタを格納する。ここでは*/dev/nicdma* へのポインタを格納しておく。

DMA バッファ領域は物理メモリの先頭付近に予約されており、DMA コントロールレジスタは MBus アドレス空間 [35:0] の終わり付近に存在する。これらを 1 度の *mmap* で取り扱うため、vm_start - vm_end で指定された領域サイズより 1 物理ページ分小さなサイズ分の DMA バッファをマップし、最後の 1 仮想ページは DMA コントロールレジスタをマップするために用いる。

---

[4] もう 1 つの方法は、システムの割り込みコントローラに直接アクセスするやりかたである。

4.1.4 割り込みハンドラ

このネットワークカードはデフォルトでは割り込みは使用しないが、DMA 完了時に割り込みをかけるかどうかをユーザーが選べるようになっている。このドライバでは、*/dev/nicdma* を *open* する際にカーネルに割り込みハンドラを登録し、*close* する際に取り除く。

こうすると、割り込みを利用しない場合でも割り込みハンドラがインストールされていることになるが、この割り込みハンドラは共有ハンドラとしてインストールされるし、自分が受け取らない割り込みは単に無視するので問題にはならない。

このドライバでは DMA 完了時の割り込みを受けて、DMA 転送を実行していたプロセスにシグナルを通知するようになっている。

割り込みはいかなるタイミングでも到着しうるため、割り込みが到着した時点で実行されているプロセスは不定である。ことによると、カーネルの実行中であったかも知れないのである。このため、*/dev/nicdma* を *open* したユーザープロセスは*/dev/nicdma* のオーナをそのプロセス ID にする。これによって、*filp-¿f_owner* の参照で DMA を実行していたプロセスが認識でき、割り込みハンドラは割り込みを受けて、DMA を実行していたプロセスに SIGIO を送ることができる。

5 おわりに

本論文では分散システム「コンピュータ・コロニー」を提案し、その高速通信ボードの機能として DMA によるブロック転送を示した。また、この高速通信ボードのサブセットとして、Linux 上で DMA 転送を用いたリモートノード間でのメモリ共有を可能にするネットワークカードを示した。さらにこのネットワークカード用の Linux デバイスドライバを作成し、Linux においてデバイスドライバが果たす役割と共に、ネットワークカードのユーザーインターフェースの実装についても述べた。

現在、コンピュータ・コロニーの通信ボードは並列環境での様々な性能評価を行なえるように、通信プロトコルの模索中である。そのうちの 1 つに DMA によるブロック転送をシステムコールを用いないで利用できるようにすることで、ソフトウェアオーバヘッドが極小のデータ転送を期待する、というものがある。本稿で示したネットワークカードは、この機能を Linux 上で実現し、その機能について考察するものである。

本稿では、このようなブロック転送の機能をユーザープログラムに解放するための方法をソフトウェアの観点から考察するため、Linux デバイスドライバの開発を行なった。この中で、比較的大きな DMA バッファを Linux において確保する方法、それを *mmap* によってユーザー空間にマッピングする方法、そしてユーザープロセスにシグナルを通知して DMA 転送の終了を伝える方法について明らかにした。

今後は、実際の並列応用においての通信方式と性能の関係をより明らかにし、コンピュータ・コロニーの主張を検証するために、オペレーティングシステム Colonia の開発を進め、通信ボードに様々な通信方式を実装することが必要である。

謝辞

本研究の機会を与えてくださった富田眞治教授に深甚な謝意を表します。また、本研究に関して適切な御指導を賜わった森眞一郎助教授、五島正裕助手に心から感謝致します。

最後に、本研究の共同研究者である山添博史氏、伊達新哉氏、Damien Le Moal 氏、杉野秀和氏、増田峰義氏、朝川達氏をはじめ日頃から御協力頂いた京都大学大学院情報学研究科通信情報システム専攻富田研究室の諸兄に心より感謝致します。