Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Departement Informatik

Institut für Computersysteme

Message Passing for Gigabit/s Networks with “Zero-Copy” under Linux

Irina Chihaia

Diploma Thesis

Technical University of Cluj-Napoca

1999

Professor: Prof. Kálmán Pusztai

Professor: Prof. Thomas M. Stricker

Assistents: Christian Kurmann and Michela Taufer

Abstract

MPICH is an OpenSource implementation of the Message Passing Interface Standard. It is built on a lower level communication layer called *Abstract Device Interface* (ADI). ADI is the central mechanism for achieving the goals of MPICH portability and performance.

Recent efforts focus on designing an optimal architecture capable of moving data between application domains and network interfaces without CPU intervention, in effect achieving "zero-copy". The implementation of communication system software with zero-copy functionality relies on some low level drivers which have access to the hardware and some higher level software services. Several zero-copy schemes have been proposed in the literature (see Bibliography).

The principle question to be answered in this thesis is, if the zero-copy strategies are a viable method to optimize MPI message passing communication over Gigabit Ethernet communication system.

In Chapter 1 we present the MPICH architecture. In the first part, there is the presentation of the ADI communication layer. The second part describes the *channel interface*, one implementation of the ADI.

Chapter 2 shortly describes the MPI principles: important MPI terms and point-to-point communication.

Some of the Linux kernel concepts are described in the Chapter 3. It contains a short presentation of the existing "zero-copy" schemes, the Ethernet characteristics, some of the most important Linux kernel internals and the idea behind a module.

Chapter 4 describes the data way in MPICH. There is a detailed explanation of the MPICH communication mechanism. We pointed out the fact that there are performed a lot of copy operations, data not being received directly into the user buffer. This chapter also contains the description of the way of TPC packets. The idea behind this description is that the Linux I/O interface is based on transfer semantics that requires copies to function correctly.

In Chapter 5 we describe the existing prototype implementation of a "zero-copy" socket interface and its restrictions. Due to the fact that it follows the principle of "User-kernel page remapping + COW" zero-copy category, its main restriction is the fact that the user buffers have to be page align. Its integration with MPICH is described here.

Chapter 6 describes the implementation of the *fast buffers (fbufs)*. This prototype implementation follows the principle of "User-kernel shared memory" zero-copy category. There is also presented the programming way using the New API.

The last Chapter contains the performance evaluation.

Contents

1 Overview of MPICH implementation **1**

1.1 Introduction 1

1.2 Architecture of MPICH 1

2 MPI principles **9**

2.1 MPI Terms 9

2.2 Point-to-Point Communication 10

3 The Linux kernel concepts **19**

3.1 Ethernet 19

3.2 Zero-Copy 20

3.3 Linux Operating System 21

3.4 Modules 26

4 The data way **27**

4.1 The data way in MPICH 27

4.2 The way of TCP packets 30

5 The existing zero-copy prototype implementation **33**

5.1 The existing zero-copy prototype 33

5.2 Integration of the existing zero-copy with MPICH 36

6 Fast Buffers (fbufs) **39**

6.1 Overview 39

6.2 Application Programming Interface 40

6.3 Programming using the New API 41
6.4 Implementation 41

7 Performance Evaluation **49**
7.1 Performance Evaluation of zero-copy implementations 49
7.2 Performance Evaluation of MPICH over Gigabit Ethernet 50

A The code of the Buffer Pool Manager **55**

List of Figures

1.1 Upper layers of MPICH 3

1.2 Lower layers of MPICH 7

3.1 Three Level Page Tables 21

3.2 A Process's Virtual Memory 23

3.3 TCP/IP Protocol Layers 24

4.1 MPICH send message 28

4.2 MPICH receive message 29

5.1 Fragmentation/Defragmentation operations 34

6.1 Fbuf allocation 40

6.2 Linux fbuf implementation 41

7.1 The TCP throughput with the existing zero-copy layer and with fbufs over Gigabit Ethernet 51

7.2 Measured throughput of MPICH without zero-copy layer over Gigabit Ethernet for non-blocking and blocking semantics 52

7.3 Mesage-passing transfer time in microseconds for MPICH transfers without zero-copy layer over Gigabit Ethernet for non-blocking and blocking semantics 53

7.4 Measured throughput of MPICH with zero-copy layer over Gigabit Ethernet for non-blocking and blocking semantics 53

List of Tables

1.1 Summary of lower-level messages 4
1.2 The bindings of lower-level messages 4
1.3 Nonblocking operations 6

2.1 Blocking send operation 12
2.2 Message envelope 12
2.3 Blocking receive operation 13
2.4 Blocking operations 13
2.5 MPICH nonblocking routines 14
2.6 MPI_WAIT function 15
2.7 MPI_IPROBE function 16
2.8 MPI_PROBE function 16
2.9 MPI_CANCEL function 16
2.10 MPID_Probe and MPID_Iprobe functions 17

4.1 Contiguous send operation 27
4.2 recv_unexpected code 30

5.1 sk_buff data structure 34
5.2 The data way in MPICH 37

6.1 The fbufs structure 42

7.1 ttcp options 50

Chapter 1

Overview of MPICH implementation

1.1 Introduction

The message-passing model of parallel computation has emerged as an expressive, efficient, and well-understood paradigm for parallel programming.

MPI (Message Passing Interface) is a specification for a standard library for message passing that was defined by the MPI Forum, a broadly based group of parallel computer vendors, library writers, and applications specialists. Multiple implementations of MPI have been developed.

MPICH is a freely available, complete implementation of the MPI specification, designed to be both portable and efficient. The "CH" in MPICH stands for "Chameleon," symbol of adaptability to one's environment and thus of portability. Chameleons are fast, and from the beginning a secondary goal was to provide as much efficiency as possible for the portability.

One of the most common parallel computing environments is a network of workstations. Many institutions use Ethernet-connected personal workstations as a "free" computational resource, and at many universities laboratories equipped with Unix workstations provide both shared Unix services for students and an inexpensive parallel computing environment for instruction. In many cases, the workstation collection includes machines from multiple vendors. Interoperability is provided by the TCP/IP standard. MPICH can run on workstations from Sun (both SunOS and Solaris), DEC, Hewlett-Packard, SGI, and IBM. The Intel 486 and Pentium compatible machines are able to join the Unix workstation family by running one of the common free implementations of Unix, such as FreeBSD, NetBSD, or Linux. MPICH can run on all of these workstations and on heterogeneous collections of them.

1.2 Architecture of MPICH

In this section we describe how the software architecture of MPICH supports the conflicting goals of portability and high performance. A detailed description can be found in [2].

The design was guided by two principles:

1. First principle was to maximize the amount of code that can be shared without compromising performance. A large amount of the code in any implementation is system independent. Implementation of most of the MPI opaque objects, including data-types, groups, attributes, and even communicators, is platform-independent. Many of the complex communication operations can be expressed portably in terms of lower-level ones.

2. Second principle was to provide a structure whereby MPICH could be ported to a new platform quickly, and then gradually tuned for that platform by replacing parts of the shared code by platform-specific code.

The central mechanism for achieving the goals of portability and performance is a specification called the Abstract Device Interface (ADI). All MPI functions are implemented in terms of the macros and functions that make up the ADI. All such code is portable. Hence, MPICH contains many implementations of the ADI, which provide portability, ease of implementation, and an incremental approach to trading portability for performance.

One implementation of the ADI is in terms of a lower level (yet still portable) interface we call the channel interface. The channel interface can be extremely small (five functions at minimum) and provides the quickest way to port MPICH to a new environment. Such a port can then be expanded gradually to include specialized implementation of more of the ADI functionality. The architectural decisions in MPICH are those that relegate the implementation of various functions to the channel interface, the ADI, or the application programmer interface (API), which in this case is MPI.

1.2.1 The Abstract Device Interface

The intention has been to allow for a range of possible functions of the device. Therefore, the design of the ADI is complex. For example, the device may implement its own message-queuing and data-transfer functions. In addition, the specific environment in which the device operates can strongly affect the choice of implementation, particularly with regard to how data is transferred to and from the user's memory space. So, if the device code runs in the user's address space, it can easily copy data to and from the user's space, while if it runs as part of the user's process (for example, as library routines on top of a simple hardware device), the "device" and the API can easily communicate, calling each other to perform services. If, on the other hand, the device is operating as a separate process and requires a context switch to exchange data or requests, then switching between processes can be very expensive, and it becomes important to minimize the number of such exchanges by providing all information needed with a single call.

Although MPI is a relatively large specification, the device-dependent parts are small. By implementing MPI using the ADI, the code can be shared among many implementations.

A message-passing ADI must provide four sets of functions:

1. specifying a message to be sent or received

2. moving data between the API and the message-passing hardware

3. managing lists of pending messages (both sent and received)

4. providing basic information about the execution environment (e.g., how many tasks are there)

The MPICH ADI provides all of these functions; however, many message-passing hardware systems may not provide list management or elaborate data-transfer abilities. These functions are emulated through the use of auxiliary routines.

The abstract device interface is a set of function definitions (which are realized as either C functions or macro definitions) in terms of which the user-callable standard MPI functions may be expressed. ADI provides the message-passing protocols that distinguish MPICH from other implementations of MPI. In particular, the ADI layer contains the code for packetizating messages and attaching header information, managing multiple buffering policies, matching posted receives with incoming messages or queuing them if necessary, and handling heterogeneous communications.

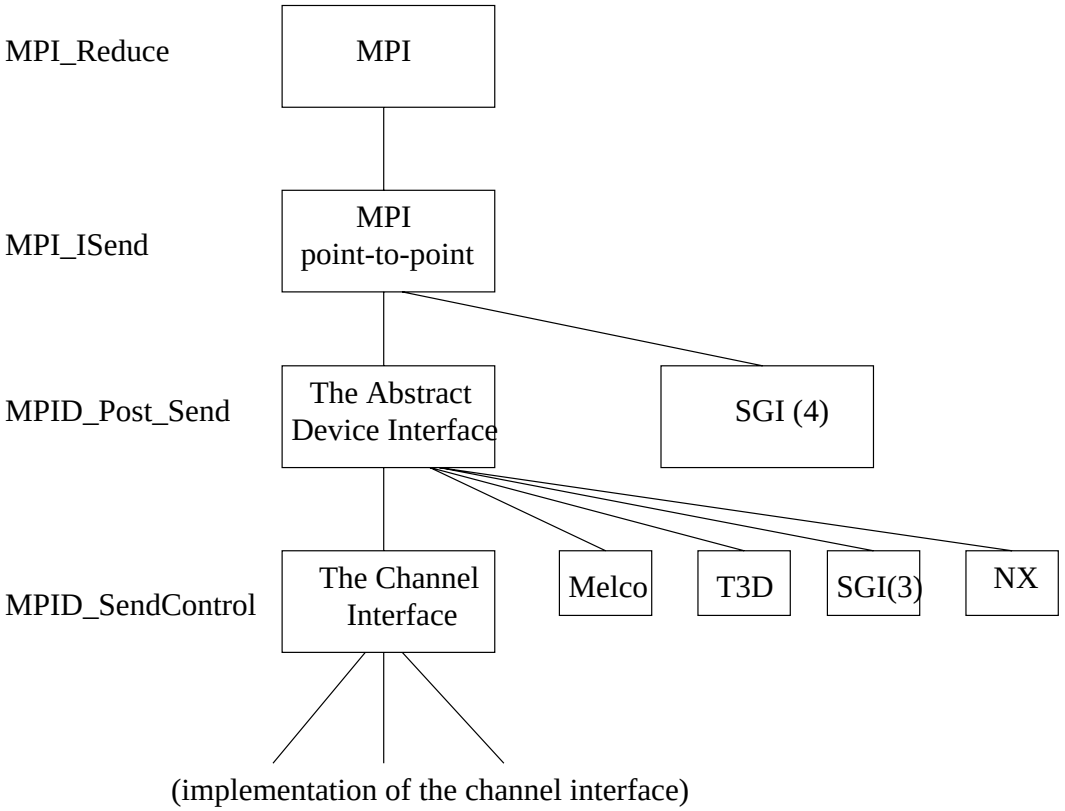

Figure 1.1: Upper layers of MPICH

A diagram of the upper layers of MPICH, including the ADI, is shown in Figure 1.1. Sample functions at each layer are displayed on the left. Without going into details now on the algorithms present in the ADI, one can expect the existence of a routine like `MPID_SendControl`, which communicates control information. The implementation of such a routine can be in

terms of a vendor's own existing message-passing system or new code for the purpose or can be expressed in terms of a further portable layer, the *channel interface*.

1.2.2 The Channel Interface

At the lowest level, what is really needed is just a way to transfer data, possibly in small amounts, from one process's address space to another's. Although many implementations are possible, the specification can be done with a small number of definitions.

The channel interface, consists of only five required functions. Three routines send and receive envelope (or control) information: `MPID_SendControl`, `MPID_ControlMsgAvail`, `MPID_RecvAnyControl`. Two routines send and receive data: `MPID_SendChannel` and `MPID_RecvFromChannel`. All these functions are described in Table 1.1.

`MPID_SendControl`	Sends a control message.
`MPID_ControlMsgAvail`	Indicates whether a control message is available.
`MPID_RecvAnyControl`	Reads the next control message. If no messages are available, blocks until one can be read.
`MPID_SendChannel`	Sends data on a particular channel.
`MPID_RecvFromChannel`	Receives data from a particular channel.

Table 1.1: Summary of lower-level messages

These operations are based on a simple capability to send data from one process to another process. No more functionality is required than what is provided by Unix in the *select*, *read*, and *write* operations. The ADI code uses these simple operations to provide the operations, such as `MPID_Post_recv`, that are used by the MPI implementation.

In Unix terms, `MPID_ControlMsgAvail` is similar to *select* (with a *fd* mask of all the file descriptors that can be read for control messages), and MPID_RecvAnyControl is like a *select* followed by a *read*, while the others are similar to *read* and *write* on the appropriate file descriptors.

Table 1.2 shows the bindings of lower-level messages.

```
void MPID_SendControl(MPID_PKT_T *pkt, int size, int dest)
int MPID_ControlMsgAvail(void)
void MPID_RecvAnyControl(MPID_PKT_T *pkt, int size, int *from)
void MPID_SendChannel(void *buf, int size, int dest)
void MPID_RecvFromChannel(void *buf, int maxsize, int from)
```

Table 1.2: The bindings of lower-level messages

In these calls, `void *buf`, `int size` is a buffer (of contiguous bytes) of size bytes. The `int dest` is the destination process (dest is the rank in `MPI_COMM_WORLD` of the destination process). The value `from` is the source of the message (also relative to `MPI_COMM_WORLD`). The value `MPID_PKT_T *pkt` is a pointer to a control message (of type `MPID_PKT_T`; this

is called a control packet or packet for short, and is defined in the file packet.h).

Buffering issues

The issue of buffering is difficult. MPICH could have defined an interface that assumed no buffering, requiring the ADI that calls this interface to perform the necessary buffer management and flow control. The reason for not making this choice is that many of the systems used for implementing the interface defined here do maintain their own internal buffers and flow controls, and implementing another layer of buffer management would impose an unnecessary performance penalty.

For correct operation of the channel interface, it is imperative that send operations do not block; that is, the completion of an `MPID_SendControl` should not require that the destination processor read the control data before the `MPID_SendControl` can complete. This requirement is because a control message is sent for any kind of MPI message, whether it is `MPI_Send`, `MPI_Isend`, `MPI_Ssend`, etc. However, in some cases, it may be more efficient to not require that the control message be delivered without requiring a matching receive. The routine `MPID_SendControlBlock` may be used for this. If this routine is not provided, then `MPID_SendControl` will be used instead. The binding for `MPID_SendControlBlock` is the same as for `MPID_SendControl`. A slight complication is the fact that the control packet itself may contain the message data, if that data is small enough. The reason for this is that it is more efficient to send the data with the control information if the data length is short enough (and doing so does not cause the sending of a control message to block).

Message ordering

Control messages between a pair of processors have to arrive in the order in which they were sent. There is no required ordering of messages sent by different processors.

When a message is sent using `MPID_SendControl` and `MPID_SendChannel`, it is guaranteed that the sending process performs the `MPID_SendChannel` after the `MPID_SendControl` for that message and before performing a `MPID_SendControl` for any other message. In other words, if the connection between two processes is a stream, the data part of a message follows immediately the control part. This applies only when the eager protocol is used. Due to the fact that we are interested only in using the eager protocol, we will not explain here the working ways for the other protocols (see Section 4.1.).

Using nonblocking operations

Nonblocking operations provide the ability to both provide greater efficiency through the overlap of computation and communication and greater robustness by allowing some data transfers to be left uncompleted until they can be received at their destination. Table 1.3 contains a short description of the nonblocking operations.

Data exchange mechanisms

The channel interface implements three different data exchange mechanisms:

1. Eager

In the *eager* protocol, data is sent to the destination immediately. If the destination is

`MPID_IRecvFromChannel(buf, size, partner, id)` Start a nonblocking receive of data from partner. The value id is used to complete this receive, and is an output parameter from this call.
`MPID_WRecvFromChannel(buf, size, partner, id)` Complete a nonblocking receive of data. Note that in many implementations, only the value of id will be used.
`MPID_RecvStatus(id)` Test for completion of a nonblocking receive.
`MPID_ISendChannel(buf, size, partner, id)` Start a nonblocking send of data to partner. The value of id is used to complete this send, and is an output parameter from this call.
`MPID_WSendChannel(buf, size, partner, id)` Complete a nonblocking send of data. Note that in many implementations, only the value of id will be used.
`MPID_TSendChannel(id)` Test for completion of a nonblocking send of data.

Table 1.3: Nonblocking operations

not expecting the data (e.g., no `MPI_Recv` has yet been issued for it), the receiver must allocate some space to store the data locally. This choice often offers the highest performance, particularly when the underlying implementation provides suitable buffering and handshakes. However, it can cause problems when large amounts of data are sent before their matching receives are posted, causing memory to be exhausted on the receiving processors. This is the default choice in MPICH.

2. Rendezvous

   In the *rendezvous* protocol, data is sent to the destination only when requested (the control information describing the message is always sent). When a receive is posted that matches the message, the destination sends to the source a request for the data. In addition, it provides a way for the sender to return the data. This choice is the most robust but, depending on the underlying system software, may be less efficient than the eager protocol. Some legacy programs may fail when run using a rendezvous protocol if an algorithm is unsafely expressed in terms of `MPI_Send`. Such a program can be safely expressed in terms of `MPI_Bsend`, but at a possible cost in efficiency. That is, the user may desire the semantics of an eager protocol (messages are buffered on the receiver) with the performance of the rendezvous protocol (no copying) but since buffer space is exhaustible and `MPI_Bsend` may have to copy, the user may not always be satisfied. MPICH can be configured to use this protocol by specifying "-use_rndv" during configuration.

3. Get

   In the *get* protocol, data is read directly by the receiver. This choice requires a method to directly transfer data from one process's memory to another. A typical implementa-

tion might use *memcpy*. This choice offers the highest performance but requires special hardware support such as shared memory or remote memory operations. In many ways, it functions like the rendezvous protocol, but uses a different set of routines to transfer the data.

MPICH includes multiple implementations of the channel interface (see Figure 1.2).

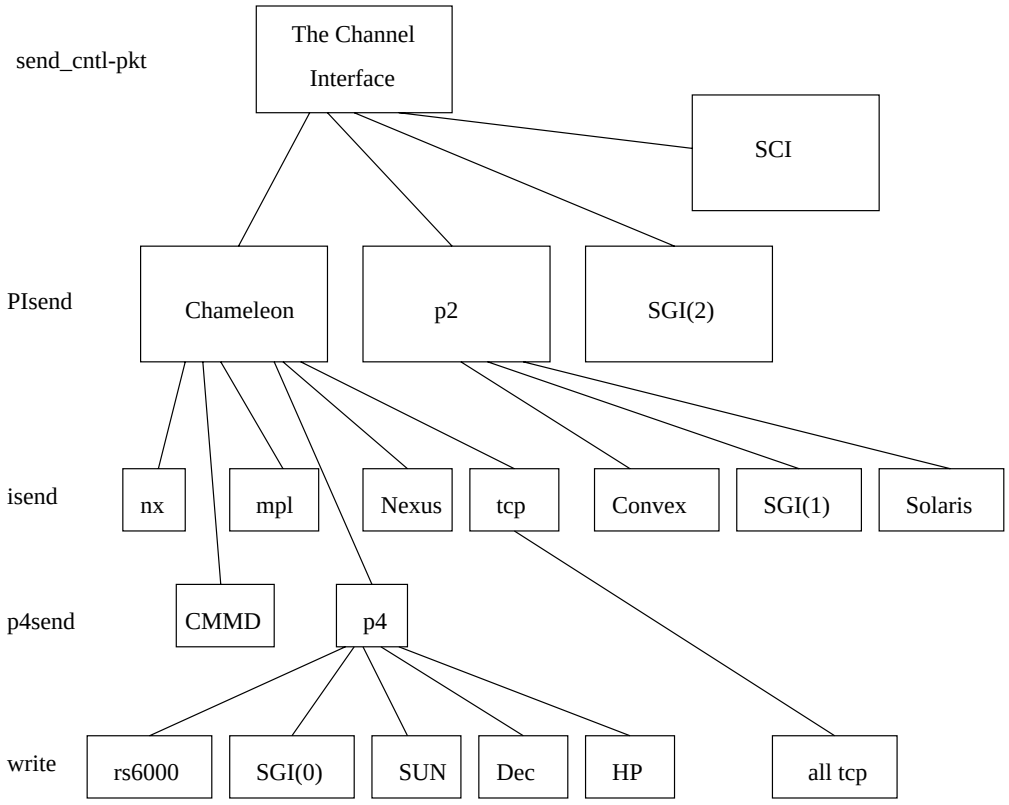

Figure 1.2: Lower layers of MPICH

Macros

The most significant implementation is the Chameleon version. By implementing the channel interface in terms of Chameleon macros, the portability is provided to a number of systems at one stroke, with no additional overhead, since Chameleon macros are resolved at compile time. Chameleon macros exist for most vendor message-passing systems, and also for p4, which in turn is portable to very many systems. The p4 implementation is only for the workstation networks.

Chapter 2

MPI principles

2.1 MPI Terms

This section explains some terms used throughout the MPI. As MPI is has large specification, we will present here only that terms we are interested for. A detailed presentation can be found at [1].

2.1.1 Semantic Terms

In this paragraph we will discuss the meaning of some MPI procedures. These MPI procedures are presented below.

1. nonblocking procedure

   A procedure is nonblocking if the procedure may return before the operation completes, and before the user is allowed to reuse resources (such as buffers) specified in the call. A nonblocking request is started by the call that initiates it, e.g., `MPI_ISEND`. A communication completes when all participating operations complete.

2. blocking procedure

   A procedure is blocking if the return from the procedure indicates the user is allowed to reuse resources specified in the call.

3. local procedure

   A procedure is local if completion of the procedure depends only on the local executing process.

4. non-local procedure

   A procedure is non-local if completion of the operation may require the execution of some MPI procedure on another process. Such an operation may require communication occurring with another user process.

5. collective procedure

   A procedure is collective if all processes in a process group need to invoke the procedure. A collective call may or may not be synchronizing. Collective calls over the same communicator must be executed in the same order by all members of the process group.

2.1.2 Data Types

MPI data types are opaque objects, array arguments, state, named constants, choice and addresses. Here we present only the opaque objects data type.

Opaque Objects

MPI manages **system memory** that is used for buffering messages and for storing internal representations of various MPI objects such as groups, communicators, data-types, etc. This memory is not directly accessible to the user, and objects stored there are **opaque**: their size and shape is not visible to the user. Opaque objects are accessed via **handles**, which exist in user space. MPI procedures that operate on opaque objects are passed handle arguments to access these objects. In addition to their use by MPI calls for object access, handles can participate in assignments and comparisons. In Fortran, all handles have type INTEGER. In C and C++, a different handle type is defined for each category of objects. In addition, handles themselves are distinct objects in C++. The C and C++ types must support the use of the assignment and equality operators. Opaque objects are allocated and deallocated by calls that are specific to each object type. The calls accept a handle argument of matching type. An opaque object and its handle are significant only at the process where the object was created and cannot be transferred to another process. This design hides the internal representation used for MPI data structures, thus allowing similar calls in C, C++, and Fortran. It also avoids conflicts with the typing rules in these languages, and easily allows future extensions of functionality. The explicit separation of handles in user space and objects in system space allows space-reclaiming and deallocation calls to be made at appropriate points in the user program. The intended semantics of opaque objects is that opaque objects are separate from one another; each call to allocate such an object copies all the information required for the object. Implementation avoids excessive copying by substituting referencing for copying. For example, a derived data-type contains references to its components, rather then copies of its components.

2.2 Point-to-Point Communication

Sending and receiving of messages by processes is the basic MPI communication mechanism. The basic point-to-point communication operations are send and receive.

One can think of message transfer as consisting of the following three phases.

1. Pull data out of the send buffer and assemble the message.
2. Transfer the message from sender to receiver.
3. Pull data from the incoming message and disassemble it into the receive buffer.

There are four communication modes:

1. standard

   In **standard** communication mode it is up to MPI to decide whether outgoing messages will be buffered. MPI may buffer outgoing messages. In such a case, the send call may complete before a matching receive is invoked. On the other hand, buffer space may be unavailable, or MPI may choose not to buffer outgoing messages, for performance reasons. In this case, the send call will not complete until a matching receive has been posted, and the data has been moved to the receiver. Thus, a send in standard mode can be started whether or not a matching receive has been posted. It may complete before a matching receive is posted. The standard mode send is **non-local**: successful completion of the send operation may depend on the occurrence of a matching receive.

2. buffer

   A **buffered** mode send operation can be started whether or not a matching receive has been posted. It may complete before a matching receive is posted. This operation is **local**, and its completion does not depend on the occurrence of a matching receive. Thus, if a send is executed and no matching receive is posted, then MPI must buffer the outgoing message, so as to allow the send call to complete.

3. synchronous

   A send that uses the **synchronous** mode can be started whether or not a matching receive was posted. However, the send will complete successfully only if a matching receive is posted, and the receive operation has started to receive the message sent by the synchronous send. A send executed in this mode is **non-local**.

4. ready

   A send that uses the **ready** communication mode may be started only if the matching receive is already posted. There is only one receive operation, which can match any of the send modes. The blocking receive operation returns only after the receive buffer contains the newly received message.

2.2.1 Blocking Send and Receive Operations

Blocking send

The syntax of the blocking send operation is given in Table 2.1.

The send buffer specified by the `MPI_SEND` operation consists of `count` successive entries of the type indicated by `datatype`, starting with the entry at address `buf`.

The data part of the message consists of a sequence of `count` values, each of the type indicated by `datatype`. The basic data-types that can be specified for message data values correspond to the basic data-types of the host language. A value of type `MPI_BYTE` consists of a byte (8 binary digits). A byte is uninterpreted and is different from a character.

```
MPI_SEND(buf, count, datatype, dest, tag, comm)
  IN buf - initial address of send buffer (choice)
  IN count - number of elements in send buffer (nonnegative integer)
  IN datatype - datatype of each send buffer element (handle)
  IN dest - rank of destination (integer)
  IN tag - message tag (integer)
  IN comm - communicator (handle)
int MPI_Send(void* buf, int count, MPI_Datatype datatype,
  int dest, int tag, MPI_Comm comm)
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
  <type> BUF(*)
  INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

Table 2.1: Blocking send operation

In addition to the data part, messages carry information that can be used to distinguish messages and selectively receive them. This information consists of a fixed number of fields, which we collectively call the message envelope. These fields are explained in Table 2.2.

`source`	determined by the identity of the message sender
`destination`	the destination of the message
`tag`	used by the program to distinguish different types of messages
`communicator`	the communicator that is used for the send operation

Table 2.2: Message envelope

A communicator specifies the communication context for a communication operation. Each communication context provides a separate "communication universe": messages are always received within the context they were sent, and messages sent in different contexts do not interfere.

A predefined communicator `MPI_COMM_WORLD` is provided by MPI. It allows communication with all processes that are accessible after MPI initialization and processes are identified by their rank in the group of `MPI_COMM_WORLD`.

Blocking receive

The syntax of the blocking receive operation is given in Table 2.3.

The receive buffer consists of the storage containing `count` consecutive elements of the type specified by `datatype`, starting at address `buf`. The length of the received message must be less than or equal to the length of the receive buffer. The selection of a message by a receive operation is governed by the value of the message envelope. A message can be received by a receive operation if its envelope matches the `source`, `tag` and `comm` values specified by the receive operation. While a receive operation may accept messages from an arbitrary sender, a send operation must specify a unique receiver. This matches a "push" communication mechanism, where data transfer is effected by the sender (rather than a "pull" mechanism, where data transfer is effected by the receiver).

```
MPI_RECV (buf, count, datatype, source, tag, comm, status)
  OUT buf - initial address of receive buffer (choice)
  IN count - number of elements in receive buffer (integer)
  IN datatype - datatype of each receive buffer element (handle)
  IN source - rank of source (integer)
  IN tag - message tag (integer)
  IN comm - communicator (handle)
  OUT status - status object (Status)
int MPI_Recv(void* buf, int count, MPI_Datatype datatype,
  int source, int tag, MPI_Comm comm, MPI_Status *status)
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS,
  IERROR)
  <type> BUF(*)
  INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM,
  STATUS(MPI_STATUS_SIZE),IERROR
```

Table 2.3: Blocking receive operation

The MPICH blocking routines correspond to the MPI routines `MPI_Send`, `MPI_Bsend`, `MPI_Rsend`, `MPI_Ssend`, and `MPI_Recv` respectively and are presented in Table 2.4.

```
MPID_SendContig(comm, buf, len, src_lrank, tag, context_id,
  dest_grank, msgrep,&error_code)
MPID_BsendContig(comm, buf, len, src_lrank, tag, context_id,
  dest_grank, msgrep, &error_code)
MPID_RsendContig(comm, buf, len, src_lrank, tag, context_id,
  dest_grank, msgrep, &error_code)
MPID_SsendContig(comm, buf, len, src_lrank, tag, context_id,
  dest_grank, msgrep, &error_code)
MPID_RecvContig(comm, buf, maxlen, src_lrank, tag, context_id,
  &status, &error_code)
```

Table 2.4: Blocking operations

2.2.2 Nonblocking communication

It is possible to improve the performance on many systems by overlapping communication and computation. A nonblocking **send start** call initiates the send operation, but does not complete it. The send start call will return before the message was copied out of the send buffer. A separate **send complete** call is needed to complete the communication, i.e., to verify that the data has been copied out of the send buffer. Similarly, a nonblocking **receive start call** initiates the receive operation, but does not complete it. The call will return before a message is stored into the receive buffer. A separate **receive complete** call is needed to complete the receive operation and verify that the data has been received into the receive buffer. The use

of nonblocking receives may also avoid system buffering and memory-to-memory copying, as information is provided early on the location of the receive buffer.

Nonblocking send start calls can use the same four modes as blocking sends: standard, buffered, synchronous and ready. These carry the same meaning. In all cases, the send start call is local: it returns immediately, irrespective of the status of other processes.

Nonblocking sends in the buffered and ready modes have a more limited impact. A nonblocking send will return as soon as possible, whereas a blocking send will return after the data has been copied out of the sender memory. The use of nonblocking sends is advantageous in these cases only if data copying can be concurrent with computation.

The message-passing model implies that communication is initiated by the sender. The communication will generally have lower overhead if a receive is already posted when the sender initiates the communication (data can be moved directly to the receive buffer, and there is no need to queue a pending send request). However, a receive operation can complete only after the matching send has occurred. The use of nonblocking receives allows one to achieve lower communication overheads without blocking the receiver while it waits for the send.

The Table 2.5 contains a summary of MPICH nonblocking routines.

```
MPID_IsendContig(comm, buf, len, src_lrank, tag, context_id,
 dest_grank, msgrep, request, &error_code)
MPID_IrsendContig(comm, buf, len, src_lrank, tag, context_id,
 dest_grank, msgrep, request, &error_code)
MPID_IssendContig(comm, buf, len, src_lrank, tag, context_id,
 dest_grank, msgpre, request, &error_code)
MPID_IrecvContig(comm, buf, maxlen, src_lrank, tag, context_id,
 request, &error_code)
```

Table 2.5: MPICH nonblocking routines

There is no nonblocking buffered send; that function can be implemented with the routines already defined, probably with little additional overhead.

The functions `MPI_WAIT` and `MPI_TEST` are used to complete a nonblocking communication. The completion of a send operation indicates that the sender is now free to update the locations in the send buffer (the send operation itself leaves the content of the send buffer unchanged). It does not indicate that the message has been received, rather, it may have been buffered by the communication subsystem. However, if a synchronous mode send was used, the completion of the send operation indicates that a matching receive was initiated, and that the message will eventually be received by this matching receive.

The completion of a receive operation indicates that the receive buffer contains the received message, the receiver is now free to access it, and that the status object is set. It does not indicate that the matching send operation has completed.

The syntax of the `MPI_WAIT` function is given in Table 2.6.

A call to `MPI_WAIT` returns when the operation identified by request is complete and

```
MPI_WAIT(request, status)
  INOUT request - request (handle)
  OUT status - status object (Status)
int MPI_Wait(MPI_Request *request, MPI_Status *status)
MPI_WAIT(REQUEST, STATUS, IERROR)
  INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
```

Table 2.6: MPI_WAIT function

contains, in status, information on the completed operation. There is a "wait-until-something-happens" routine called `MPID_DeviceCheck`. This can be null but can be used to allow `MPI_Wait` to wait more passively (for example, with a select) rather than spinning on tests. It takes one argument of type `MPID_BLOCKING_TYPE`; it can be used for both blocking and nonblocking checks of the device. The routine `MPID_DeviceCheck` may be used to wait for something to happen. This routine may return at any time; it must return if the is_complete value for any request is changed during the call. For example, if a request would be completed only when an acknowledgment message or data message arrived, then `MPID_DeviceCheck` could block until a message (from anywhere) arrived.

2.2.3 Queue Information

We will explain here the `MPI_PROBE` and `MPI_IPROBE` operations. These operations allow incoming messages to be checked for, without actually receiving them. The user can then decide how to receive them, based on the information returned by the probe (basically, the information returned by status). In particular, the user may allocate memory for the receive buffer, according to the length of the probed message. The MPI implementation of `MPI_PROBE` and `MPI_IPROBE` needs to guarantee progress: if a call to `MPI_PROBE` has been issued by a process, and a send that matches the probe has been initiated by some process, then the call to `MPI_PROBE` will return, unless the message is received by another concurrent receive operation (that is executed by another thread at the probing process). Similarly, if a process busy waits with `MPI_IPROBE` and a matching message has been issued, then the call to `MPI_IPROBE` will eventually return flag true unless the message is received by another concurrent receive operation. The `MPI_CANCEL` operation allows pending communications to be canceled. This is required for cleanup. Posting a send or a receive ties up user resources (send or receive buffers), and a cancel may be needed to free these resources gracefully.

The syntax of the `MPI_IPROBE` function is given in Table 2.7.

`MPI_IPROBE` returns the flag true if there is a message that can be received and that matches the pattern specified by the arguments `source`, `tag`, and `comm`. The call matches the same message that would have been received by a call to `MPI_RECV` executed at the same point in the program, and returns in `status` the same value that would have been returned by `MPI_RECV`. Otherwise, the call returns the flag false, and leaves status undefined. A subsequent receive executed with the same context, and the source and tag returned in status by `MPI_IPROBE` will receive the message that was matched by the probe, if no other intervening receive occurs after the probe. It is not necessary to receive a message immediately after it has

```
MPI_IPROBE(source, tag, comm, flag, status)
  IN source - source rank, or MPI_ANY_SOURCE (integer)
  IN tag - tag value or MPI_ANY_TAG (integer)
  IN comm - communicator (handle)
  OUT flag - (logical)
  OUT status - status object (Status)
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,
  MPI_Status *status)
MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)
  LOGICAL FLAG
  INTEGER SOURCE, TAG, COMM,
  STATUS(MPI_STATUS_SIZE), IERROR
```

Table 2.7: MPI_IPROBE function

been probed for, and the same message may be probed for several times before it is received.

`MPI_PROBE` behaves like `MPI_IPROBE` except that it is a blocking call that returns only after a matching message has been found (see Table 2.8).

```
MPI_PROBE(source, tag, comm, status)
  IN source - source rank, or MPI_ANY_SOURCE (integer)
  IN tag - tag value, or MPI_ANY_TAG (integer)
  IN comm - communicator (handle)
  OUT status - status object (Status)
int MPI_Probe(int source, int tag, MPI_Comm comm,
  MPI_Status *status)
MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)
  INTEGER SOURCE, TAG, COMM,
  STATUS(MPI_STATUS_SIZE), IERROR
```

Table 2.8: MPI_PROBE function

As for `MPI_IPROBE`, a call to `MPI_PROBE` will match the message that would have been received by a call to `MPI_RECV` executed at the same point.

The syntax of the `MPI_CANCEL` function is given in Table 2.9.

```
MPI_CANCEL(request)
  IN request - communication request (handle)
int MPI_Cancel(MPI_Request *request)
MPI_CANCEL(REQUEST, IERROR)
  INTEGER REQUEST, IERROR
```

Table 2.9: MPI_CANCEL function

A call to `MPI_CANCEL` marks for cancellation a pending, nonblocking communication

operation (send or receive). The cancel call is local. It returns immediately, possibly before the communication is actually canceled. The successful cancellation of a buffered send frees the buffer space occupied by the pending message. Either the cancellation succeeds, or the communication succeeds, but not both. If a send is marked for cancellation, either the send completes normally, in which case the message sent was received at the destination process, or the send is successfully canceled, in which case no part of the message was received at the destination. Then, any matching receive has to be satisfied by another send. On the other hand if a receive is marked for cancellation, either the receive completes normally, or the receive is successfully canceled, in which case no part of the receive buffer is altered. Moreover, any matching send has to be satisfied by another receive. Cancel can be an expensive operation that should be used only exceptionally. If a send operation uses an "eager" protocol (data is transferred to the receiver before a matching receive is posted), the cancellation of this send requires communication with the intended receiver in order to free allocated buffers. On some systems this may require an interrupt to the intended receiver.

The **corresponding ADI functions** are `MPID_Probe` (blocking test) and `MPID_Iprobe` (nonblocking test). Since these perform message matching in the same way as the receive routines do, they have similar argument lists (see Table 2.10).

```
MPID_Probe(comm, tag, context_id, src_lrank,
  &error_code, &status)
MPID_Iprobe(comm, tag, context_id, src_lrank,
   &flag, &error_code, &status)
```

Table 2.10: MPID_Probe and MPID_Iprobe functions

Note that `MPID_Probe` and `MPID_Iprobe` may need both relative rank and global rank, depending on how the message is tested for. However, since the communicator is passed to the device, it is as easy for the device to convert relative rank to absolute rank as it is for the MPI implementation. This is the same as for the corresponding `MPID_Recvxxx` routines.

Chapter 3

The Linux kernel concepts

3.1 Ethernet

The term Ethernet refers to the family of local area network (LAN) implementations that include three principal categories.

1. Ethernet

   Ethernet is a broadcast network. In other words, all stations see all frames, regardless of whether they represent an intended destination. Each station must examine received frames to determine if the station is a destination. If so, the frame is passed to a higher protocol layer for appropriate processing. Ethernet provides services corresponding to Layers 1 (physical) and 2 (link layer) of the OSI reference model. Ethernet is implemented in hardware. Typically, the physical manifestation of these protocols is either an interface card in a host computer or circuitry on a primary circuit board within a host computer. Ethernet offers 10 Mbps of raw-data bandwidth.

2. 100-Mbps Ethernet

   100-Mbps Ethernet is a high-speed LAN technology that offers increased bandwidth to desktop users in the wiring center, as well as to servers and server clusters in data centers. 100BaseT is the IEEE specification for the 100-Mbps Ethernet implementation over unshielded twisted-pair (UTP) and shielded twisted-pair (STP) cabling.

3. Gigabit Ethernet

   Gigabit Ethernet is an extension of the IEEE 802.3 Ethernet standard. Gigabit Ethernet offers 1000 Mbps of raw-data bandwidth while maintaining compatibility with Ethernet and Fast Ethernet network devices. Gigabit Ethernet provides for new, full-duplex operating modes for switch-to-switch and switch-to-end-station connections. It also permits half-duplex operating modes for shared connections by using repeaters and CSMA/CD. Furthermore, Gigabit Ethernet uses the same frame format, frame size, and management objects used in existing IEEE 802.3 networks. In general, Gigabit Ethernet is expected to initially operate over fiber-optic cabling but will be implemented over unshielded twisted-pair (UTP) and coaxial cabling as well.

Ethernet has survived as an essential media technology because of its tremendous flexibility and its relative simplicity to implement and understand. Although other technologies have been touted as likely replacements, network managers have turned to Ethernet and its derivatives as effective solutions for a range of campus implementation requirements. To resolve Ethernet's limitations, innovators (and standards bodies) have created progressively larger Ethernet pipes. Critics might dismiss Ethernet as a technology that cannot scale, but its underlying transmission scheme continues to be one of the principal means of transporting data for contemporary campus applications.

3.2 Zero-Copy

Recent efforts have been focused on designing an optimal architecture called "zero-copy" capable of moving data between application domains and network interfaces without CPU intervention. The overhead in networking software can be broken up into *per-packet* and *per-byte* costs. The per-packet cost is roughly constant for a given network protocol, regardless of the packet size, whereas the per-byte cost is determined by the data copying and the checksumming overhead. In general, the communication link imposes an upper bound limit on the packet size that can be accepted (called maximum transfer unit or MTU). On Ethernet MTU has the value of 1500 Bytes and the communication overhead is dominated by the per-packet cost.

Several zero-copy schemes have been proposed in the literature. In the following, we will classify the approaches into four categories, and briefly describe them.

1. User accessible interface memory

   One scenario with minimal data transfer overhead is one in which the network interface memory is accessible and pre-mapped into the user and kernel address space. Data never needs to traverse the memory bus until it is accessed by the application. Unfortunately, this requires complicated hardware support and substantial software changes.

2. Kernel-network shared memory

   This scheme lets the operating system kernel manage the interface and uses direct memory access (DMA) or programmed I/O (PIO) to move data between interface memory and application buffers.

3. User-kernel shared memory

   This scheme defines a new set of application programming interfaces (APIs) with shared semantics between the user and kernel address space and uses DMA to move data between the shared memory and network interface. One proposal in this category is called *fast buffers (fbufs)* described by Druschel and Peterson. It uses per-process buffer pool that is pre-mapped in both the user and kernel address spaces, thus eliminating the user-kernel data copy.

4. User-kernel page re-mapping + COW

   This scheme uses DMA to transfer data between interface memory and kernel buffers, and re-maps buffers by editing the MMU table to give the appearance of data transfer.

By combining it with COW (copy-on-write) technique on the transmit side, it preserves the copy semantics of the traditional socket interface.

3.3 Linux Operating System

Linux is made up of a number of functionally separate pieces that, together, comprise the operating system. One obvious part of Linux is the kernel itself; but even that would be useless without libraries or shells.

3.3.1 Memory management

One of the basic tricks of any OS is the ability to make a small amount of physical memory behave like rather more memory. This apparently large memory is known as virtual memory. The system divides the memory into easily handled pages.

Each process in the system has its own virtual address space. These virtual address spaces are completely separate from each other and so a process running one application cannot affect another. These virtual addresses are converted into physical addresses by the processor based on the info held in a set of tables maintained by the OS. Linux assumes that there are three levels of Page Tables. To translate a virtual address into a physical one, the processor must take the contents of each level field, convert it into an offset into the physical page containing the Page Table and read the page frame number of the next level of Page Table. These is repeated three times until the page frame number of the physical page containing the virtual address is found. Now , the final field in the virtual address, the byte offset, is used to find the data inside the page (see Figure 3.1).

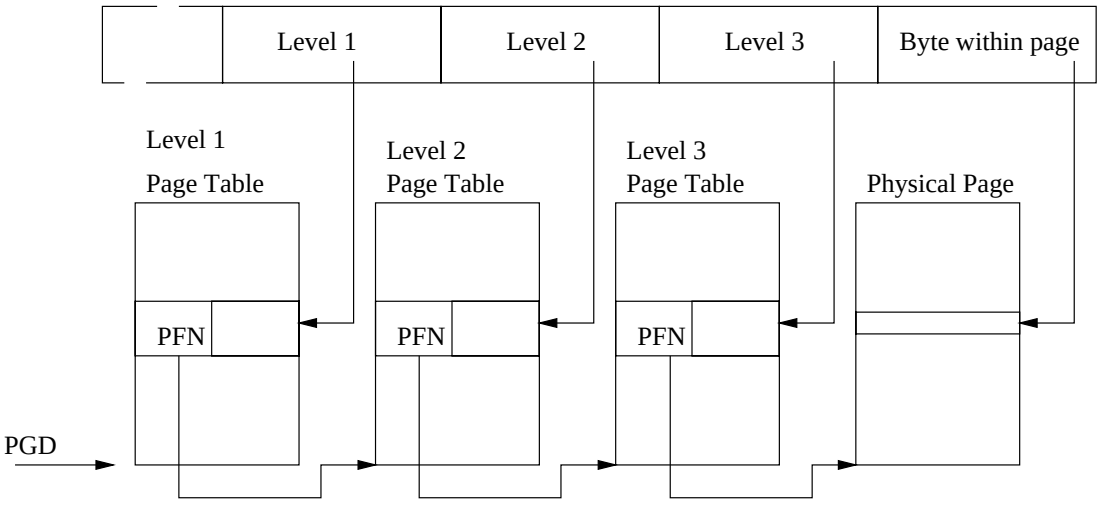

Figure 3.1: Three Level Page Tables

The mechanisms and data structures used for **page allocation and deallocation** are perhaps the most critical in maintaining the efficiency of the virtual memory subsystem. The

kernel manages the system's physical memory, which is available only in page-sized chunks. This fact leads to a page-oriented allocation technique. Linux addresses the problem of managing *kmalloc*'s needs by administering a page pool, so that pages can be added or removed from the pool easily. For versions 2.1.38 and newer the implementation of *kmalloc* is replaced with a completely new one. The 2.0 implementation of memory allocation can be seen in *mm/kmalloc.c* while the new one lives in *mm/slab .c* (the implementation deviates from Bonwick's papers describing the slab allocator).

Memory mapping is used to map image and data files into a process address space. In memory mapping, the contents of a file are linked directly into the virtual address space of a process.

Every process virtual memory is represented by an `mm_struct` data structure. This contains info about the image that is currently executing and also has pointers to a number of `vm_area_struct` data structures, each representing an area of virtual memory within this process. Each `vm_area_struct` data structure describes the start and the end of virtual memory, the process access rights to that memory and a set of operations for that memory (see Figure 3.2).

When an executable image is mapped into a process virtual address a set of `vm_area_struct` data structures is generated. Each `vm_area_struct` data structure represents a part of the executable image; the executable code, initialized data (variable), uninitialized data and so on.

The process's set of `vm_area_struct` data structures is accessed repeatedly by the Linux kernel as it creates new areas of virtual memory for the process and as it fixed up references to virtual memory not in the system's physical memory. This makes the time that it takes to find the correct `vm_area_struct` critical to the performance of the system. To speed up this accesses, Linux arranges the `vm_area_struct` data structures into an AVL (Andelson-Velskii and Landis) tree. This tree is arranged so that each `vm_area_struct` has a left and a right pointer to its neighboring `vm_area_struct` structure. The left pointer points to node with a lower starting virtual address and the right pointer points to a node with a higher starting virtual address.

3.3.2 Network Devices

Device drivers make up a major part of the Linux kernel. They control the interaction between the OS and the hardware device that they are controlling.

A network device is, so far as Linux's network subsystem is concerned, an entity that sends and receives packets of data. This is normally a physical device such as Ethernet card. The device uses standard networking support mechanisms to pass received data up to the appropriate protocol layer. All network data (packets) transmitted and received are represented by `sk_buff` data structures, these are flexible data structures that allow network protocol headers to be easily added and removed.
The device data structure contains info about the network device:

Name The names of the network devices are standard, each name representing the type of device that it is.

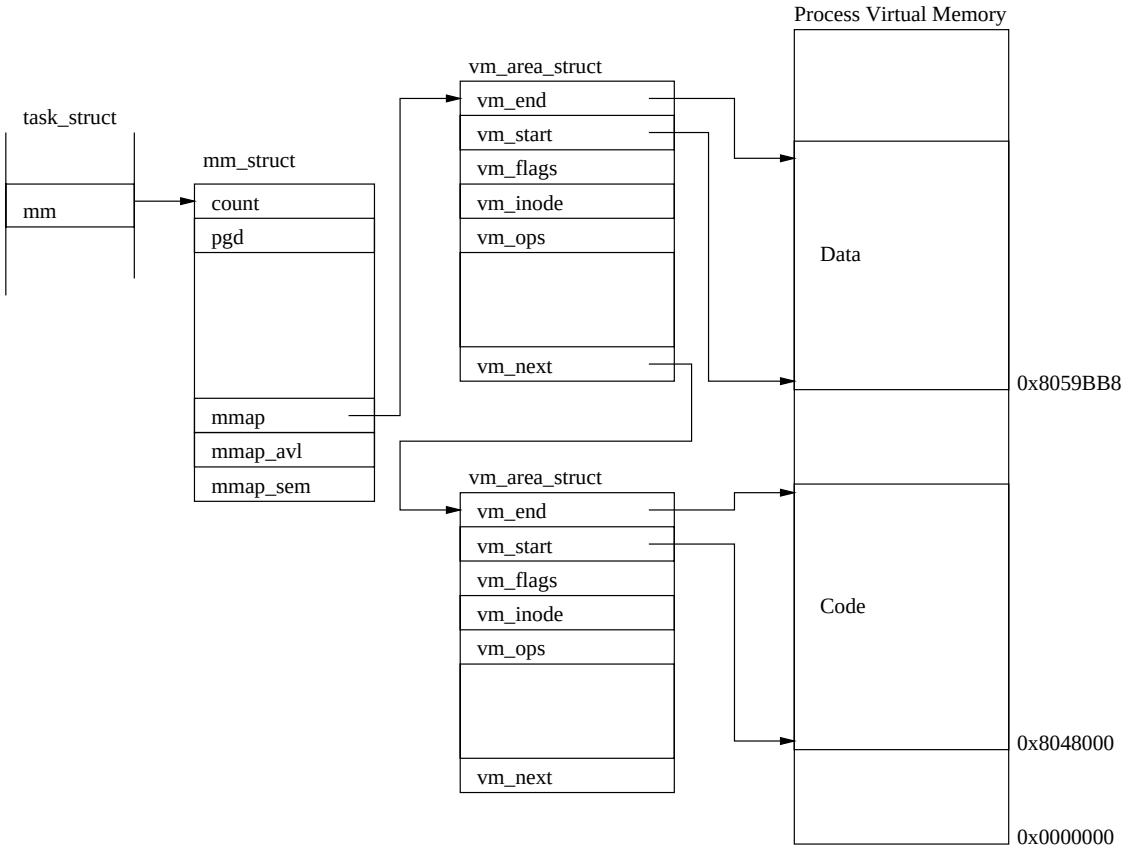

Figure 3.2: A Process's Virtual Memory

Bus information This is info that the device driver needs in order to control the device. The *irq* number is the interrupt that this device is using. The *base address* is the address of any of the device's control and status registers in I/O memory. The DMA *channel* is the DMA channel number that this network device is using. All these info are set at boot time as the device is initialized.

Interface flags These describe the characteristics and abilities of the network device.

Protocol Information Each device describes how it can may be used by the network protocol layers. One of the most important protocol info is MTU. MTU is the size of the largest packet that this can transmit not including any link layer headers that it needs to add. This maximum is used by the protocol layers, for example IP, to select suitable packet sizes to send.

Packet Queue This is the queue of `sk_buff` packets queued waiting to be transmitted on this network device.

3.3.3 Networks

This section describes how Linux supports the network protocols known collectively as TCP/IP. The TCP/IP protocols were designed to support communications between computers connected to the ARPANET, an American research network funded by the US government.

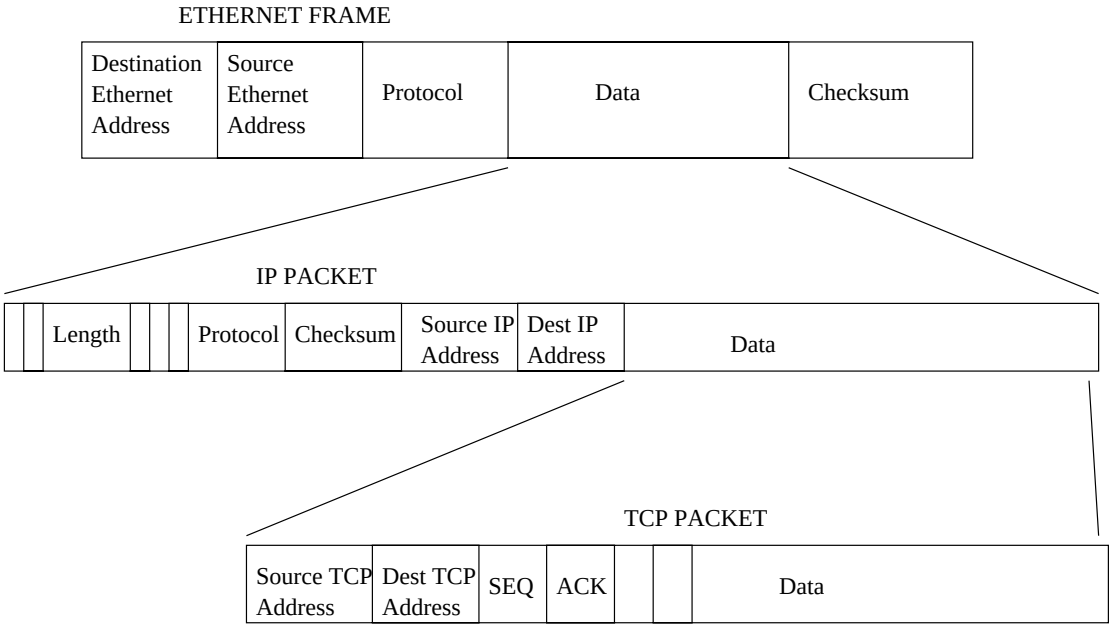

Figure 3.3: TCP/IP Protocol Layers

The IP protocol is a transport layer that is used by the other protocols to carry their data. Whenever you connect to another machine, its IP address is used to exchange data with that machine. This data is contained in IP packets each of which have an IP header containing the IP addresses of the source and destination machine's IP addresses, a checksum and other useful information. The checksum is derived from the data in the IP packet and allows the receiver of IP packets to tell if the IP packet was corrupted during transmission. The data transmitted by an application may have been broken down into smaller packets which are easier to handle. The size of the IP data packets varies depending on the connection media. The destination host must reassemble the data packets before giving the data to the receiving application.

The Transmission Control Protocol (TCP) is a reliable end to end protocol that uses IP to transmit and receive its own packets. Just as IP packets have their own header, TCP has its own header. TCP is a connection based protocol where two networking applications are connected by a single, virtual connection even though there may be many subnetworks, gateways and routers between them. TCP reliably transmits and receives data between the two applications and guarantees that there will be no lost or duplicated data. When TCP transmits its packet using IP, the data contained within the IP packet is the TCP packet itself. The IP layer on each communicating host is responsible for transmitting and receiving IP packets.

When IP packets are received, the receiving IP layer must know which upper protocol layer to give the data contained in this IP packet to. To facilitate this every IP packet header has

a byte containing a protocol identifier. When TCP asks the IP layer to transmit an IP packet, that IP packet's header states that it contains a TCP packet. The receiving IP layer uses that protocol identifier to decide which layer to pass the received data up to, in this case the TCP layer.

This layering of protocols does not stop with TCP and IP. The IP protocol layer itself uses many different physical media to transport IP packets to other IP hosts. These media may themselves add their own protocol headers. One such example is the Ethernet layer. An Ethernet network allows many hosts to be simultaneously connected to a single physical cable. Every transmitted Ethernet frame can be seen by all connected hosts and so every Ethernet device has a unique address. These unique addresses are built into each Ethernet device when they are manufactured. As Ethernet frames can carry many different protocols (as data) they, like IP packets, contain a protocol identifier in their headers. This allows the Ethernet layer to correctly receive IP packets and to pass them onto the IP layer (see Figure 3.3).

Socket Buffers

One of the problems of having many layers of network protocols, each one using the services of another, is that each protocol needs to add protocol headers and tails to data as it is transmitted and to remove them as it processes received data.

The Linux solution to this problem is to make use of socket buffers or `sk_buffs` to pass data between the protocol layers and the network device drivers. `sk_buffs` contain pointer and length fields that allow each protocol layer to manipulate the application data via standard functions.

Receiving IP Packets

When a network device receives packets from its network it must convert the received data into `sk_buff` data structures. These received `sk_buff`'s are added onto the backlog queue by the network drivers as they are received. The network bottom half is flagged as ready to run as there is work to do. When the network bottom half handler is run by the scheduler it processes any network packets waiting to be transmitted before processing the backlog queue of `sk_buff`'s determining which protocol layer to pass the received packets to.

Sending IP Packets

Packets are transmitted by the applications exchanging data or else they are generated by the network protocols as they support established connections or connections being established. Whichever way the data is generated, an `sk_buff` is built to contain the data and various headers are added by the protocol layers as it passes through them.

The `sk_buff` needs to be passed to a network device to be transmitted. This device will be established based on the best route for packet. For every IP packet transmitted, IP uses the routing tables to resolve the route for the destination IP address.

Data fragmentation

Every network device has a maximum packet size and it cannot transmit or receive a data packet bigger than this. The IP protocol allows for this and will fragment data into smaller units to fit into the packet size that the network device can handle. The IP protocol header includes a fragment field which contains a flag and the fragment offset.

When an IP packet is ready to be transmitted, IP finds the network device to send the IP packet out on. This device is found from the IP routing tables. Each device has a field describing its maximum transfer unit (in bytes), this is the MTU field. If the device's MTU is smaller than the packet size of the IP packet that is waiting to be transmitted, then the IP packet must be broken down into smaller (MTU sized) fragments. Each fragment is represented by a `sk_buff`; its IP header marked to show that it is a fragment and what offset into the data this IP packet contains. The last packet is marked as being the last IP fragment. If, during the fragmentation, IP cannot allocate an `sk_buff`, the transmit will fail.

Receiving IP fragments is a little more difficult than sending them because the IP fragments can be received in any order and they must all be received before they can be reassembled. Each time an IP packet is received it is checked to see if it is an IP fragment. The first time that the fragment of a message is received, IP creates a new `ipq` data structure, and this is linked into the `ipqueue` list of IP fragments awaiting recombination. As more IP fragments are received, the correct `ipq` data structure is found and new `ipfrag` data structure is created to describe this fragment. Each `ipq` data structure uniquely describes an fragmented IP receive frame with its source and destination IP addresses, the upper layer protocol identifier and the identifier for this IP frame. When all the fragments have been received, they are combined into a single `sk_buff` and passed up to the next protocol level to be processed. Each `ipq` contains a timer that is restarted each time a valid flag fragment is received. If this timer expires, the `ipg` data structure and its `ipfrag`'s are discarded and the message is presumed to have been lost in transit. It is then up to the higher level protocols to retransmit the message.

3.4 Modules

Linux allows the programmer to dynamically load and unload components of the OS as he need them. Linux modules are lumps of code that can be dynamically linked into the kernel and removed when they are no longer needed. Mostly Linux kernel modules are device drivers, pseudo-device drivers such as network drivers, or file-systems. Dynamically loading code as it is needed is attractive as it keeps the kernel size to a minimum and makes the kernel very flexible. Once a Linux module has been loaded it is as much part of the kernel as normal kernel code. It has the same rights and responsibilities as kernel code.

So that modules can use the kernel resources that they need, they must be able to find them. The kernel keeps a list of all of the kernel' s resources in the kernel symbol table so that it can resolve references to those resources from the modules as they are loaded. Linux allows module stacking, this is where one module requires the services of another module. As each module is loaded, the kernel modifies the kernel symbol table, adding to it all the resources or symbols exported by the newly loaded module. This means that, when the next module is loaded, it has access to the services of the already loaded modules. When an attempt is made to unload a module, the kernel needs to known that the module is unused and it needs some way of notifying the module that it is about to be unloaded. That way the module will be able to free up any system resources that it has allocated. When the module is unloaded, the kernel removes any symbol that that module exported into the kernel symbol table.

Chapter 4

The data way

4.1 The data way in MPICH

The MPI standard is designed to give an MPI implementation freedom in choosing exactly how and when data should be moved from one process to another. Since no single choice of protocol is optimal, a combination of transmission protocols is appropriate. The protocol between the sender and the receiver negotiates the best way of a data transfer and this is based on the message size. This negotiation takes place at the Channel Interface level. We will consider the dimension of a page as being of 4096 Byte like in the IBM/PC case. For short messages - from 0 to 16384 Byte (4 pages), MPICH uses the *short* protocol (data is delivered within the message envelope). For long messages - from 16384 Byte (4 pages) to 131072 Byte (32 pages), it is used the *eager* protocol (data is sent to the destination immediately). The *rendezvous* protocol (data is sent to the destination only when requested) is used for dealing with very long messages - up to 131072 Byte (32 pages).

4.1.1 Send operation

In MPICH the messages are sent with `MPI_Send` function (mpich/src/pt2pt/send.c) and the protocol negotiation is defined in mpich/mpid/ch_p4/adi2send.c.

Table 4.1 shows the contiguous send operation.

```
/* Choose the function based on the message length in bytes */
if (len < dev->long_len)
  fcn = dev->short_msg->send;
else if (len < dev->vlong_len && MPID_FLOW_MEM_OK(len,dest_grank))
  fcn = dev->long_msg->send;
else
  fcn = dev->vlong_msg->send;
```

Table 4.1: Contiguous send operation

Messages are composed of two parts: a *control part* (or message envelope), containing useful informations, and the *data part*, containing the actual data.

In order to reduce the latency of **short messages**, data is sent with the control part of the message instead of being sent in a separate data part (see Figure 4.1.). This leads to a copy operation (in mpich/mpid/ch_p4/chshort.c) but avoids some inefficient transfers across the PCI bus. The copy operation takes place in the function `MPID_CH_Eagerb_send_short` and after this, the message is passed to the function `MPID_SendControlBlock`.

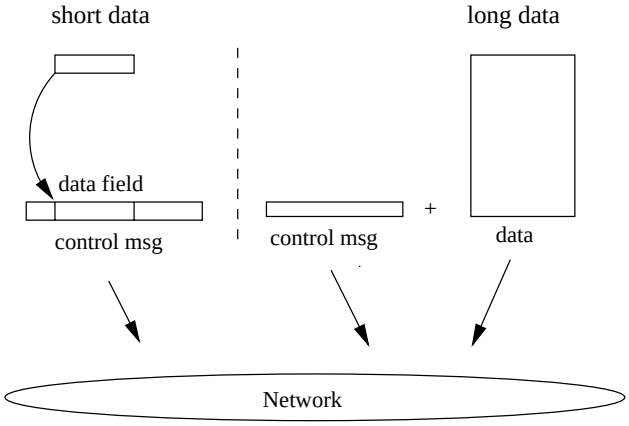

Figure 4.1: MPICH send message

For **long messages**, firstly, it is sent the control message (constructed here) using the function `MPID_SendControlBlock` and after that, the corresponding data by making use of the function `MPID_SendChannel` defined in mpich/mpid/ch_p4/chbeager.c. In this case data is sent without copies, directly from the user buffer (see Figure 4.1).

In case of **very long messages**, firstly, some request info are stored and after this, the corresponding control message is sent to the destination. All these operations take place in the function `MPID_CH_Rndvb_isend` defined in mpich/mpid/ch_p4/chbrndv.c.

We will deal only with short and long messages. In both cases the `MPID_SendControlBlock` function is used. It calls `PIbsend` function (/mpich/mpid/ch2/channel.h). Since we are considering only point-to-point communication we have to mention that at the end data is sent on the network by calling the function `net_send` (defined in mpich/mpid/ch_p4/p4/lib/p4_sock_util.c). This function is called by `socket_send` (mpich/mpid/ch_p4/p4/lib/p4_sock_sr.c). The function `net_send` makes use of a variable called “trial_size’. This variable is set large enough - 65536 Byte (16 pages) - to let us do long writes.

For modes of operation that involve packing/unpacking operations of messages into buffers, the case of heterogeneous networks and of gather- and scatter-operations, MPICH performs a copy operation.

4.1.2 Receive operation

On the receiving part, the corresponding MPI function is `MPI_Recv` (mpich/src/pt2pt/recv.c). For receives, two different situations are possible: the message is expected, and the message is unexpected (see Figure 4.2).

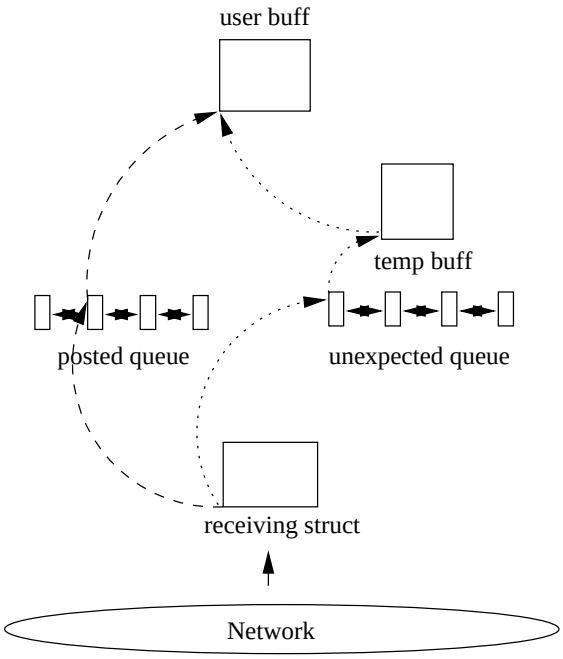

Figure 4.2: MPICH receive message

At the upper lever (the receiving user buffer is defined), when `MPI_Recv` function is called, MPICH will make use of `MPID_IrecvContig` function. `MPID_IrecvContig` (defined in mpich/mpid/ch_p4/adi2recv.c) checks to see if the message has already been received, by calling `MPID_Search_unexpected_queue_and_post`. We cannot have any thread receiving a message while checking the queues. In case we enqueue the message, we set the fields that will need to be valid before calling this routine (this is extra overhead only in the case that the message was unexpected, which is already the higher-overhead case). All routines for managing message queues can be found in mpich/mpid/ch_p4/queue.c.

The function `MPID_Search_unexpected_queue_and_post` first checks the unexpected queue by calling `MPID_Search_unexpected_queue` and then post the receive if a matching value is not first found in the unexpected queue. The output value of `MPID_Search_unexpected_queue` is non-null if an element is found and in this case the entry is removed from the unexpected queue by calling the function `MPID_SBfree`. The post operation is made by calling the function `MPID_Enqueue` which has as arguments the posted queue and the address of the receiving buffer (allocated by the user). This function prepares a corresponding entry into the posted receive queue for the message that has to be received. When the message will came, it will be received in the user allocated buffer (provided by `MPI_Recv` function).

At the lower level (data is received from network) instead of checking for whether a message is available, MPICH tries and receives a message with "check-device". The "check-device" code uses a lookup table based on the packet type field (small integer) and selects the routine that matches the kind of send that was used. The table is initialized when the device starts up.

The **most important fact** is that at lowest level, in function `socket_recv`, data is received into a statically allocated structure of type `slave_listener_msg` (mpich/mpid/ch_p4/p4_sock_sr.c). At the upper levels the received data is copied into the corresponding field from posted or unexpected queues. This copy operation takes place in `PIbrecv` defined in mpich/mpid/ch_p4/chdef.h.

When a message arrives the device calls `MPID_Msg_arrived` function. This function makes use of `MPID_Search_posted_queue`. In case of posted receives, at the moment of the message arrival, the receiving device data structure has already been created. The interrupt calls device, data is transfered and the receive is marked as completed. In case of finding an unexpected message, the `recv_unexpected` code looks like in Table 4.2:

```
recv_unexpected
  Save matched data (tag, source)
  Check for message data available (short or eager delivery)
  Save functions (test,wait,push)
  if (req->push) (req->push)(req)
```

Table 4.2: recv_unexpected code

When an `MPI_Recv` call matches a message from the unexpected queue, the device transfers the message into the corresponding user space. At this moment, data is copied from the receiving handle (allocated by `MPID_RecvAlloc`) at the requested user address (provided only in this moment). This copy operation takes place in `MPID_CH_Eagerb_save` defined in mpich/mpid/ch_p4/chbeager.c

4.2 The way of TCP packets

Linux's networking implementation is modeled on 4.3 BSD in that it supports BSD sockets and the full range of TCP/IP networking. A network interface deals with packet transmission and reception at the kernel level. Interaction between a network driver and the kernel proper deals with one packet at a time; this allows protocol issues to be hidden neatly from the driver and the physical transmission to be hidden from the protocol.

When the sender process calls the function `write`, this will call the kernel function `sock_write`. `sock_write` calls the function `sock_sendmsg` with the proper parameters set. In the file net/socket.c there are defined all functions dealing with the functional part of the socket. When the function `tcp_do_sendmsg` is called by `sock_write`, it will allocate an `sk_buff` data structure (in the kernel space). This `sk_buff` data structure is used to pass data between the protocol layers and the network device drivers. In function

`tcp_do_sendmsg` data from the user buffer will be copied into the `sk_buff` data structure (kernel space) by making use of the function `csum_and_copy_from_user`. The file net/ipv4/tcp.c holds the definition of the function `tcp_do_sendmsg`.

The resulted `sk_buff` data structure is forward sent by calling the `tcp_send_skb` function, the TCP header being constructed here. Next, the TCP packet is passed to the IP layer by calling `ip_queue_xmit` function. After the IP header is built, the packet is sent to `dev_queue_xmit`. Finally, the function `start_xmit` of the device driver is called and the packet is sent on the network.

When a network device receives packets from its network it must convert the received data into `sk_buff` data structures. These received `sk_buff`'s are added onto the backlog queue by the network drivers as they are received. The network bottom half is flagged as ready to run as there is work to do. When the network bottom half handler is run by the scheduler, it processes any network packets waiting to be transmitted, before processing the backlog queue of `sk_buff`'s determining which protocol layer to pass the received packets to. The `sk_buff`'s are sent to the `ip_rcv` function defined in net/ipv4/ip_input.c. The defragmentation operation takes place in function `ip_defrag`. The TCP layer will receive packets by making use of the function `get_tcp_sock`. Data has to be copied from the `sk_buff` data structure to the corresponding user space that is provided by the function `read`. This copy operation takes place in function `tcp_recv_msg`.

Chapter 5

The existing zero-copy prototype implementation

5.1 The existing zero-copy prototype

The existing prototype implementation of a zero-copy socket interface follows the principle of "User-kernel page re-mapping + COW" zero-copy category. This scheme uses the DMA to transfer data between interface memory and kernel buffers, and re-maps buffers by editing the MMU table to give the appearance of data transfer. By combining it with the COW (copy-on-write) technique on the transmit side, it preserves the copy semantics of the traditional socket interface.

This approach has the following **drawbacks**. All the buffers involved must align on page boundaries, and occupy an integral number of MMU pages. The packets must be large enough to cover a whole page of data. In order to apply page re-mapping, user payload of all the packets must contain an integral number of pages. The network driver software has to accurately predict the size of protocol headers to skip. This requires that the header of TCP/IP protocol has a fixed length, so no IP options are allowed.

Due to the fact that packets must be large enough to cover a whole page of data, the `sk_buff` data structure must be changed. In order to preserve the TCP/IP compatibility, a new variable `zc_data` takes part in the `sk_buff` data structure. The zero-copy data buffer is allocated by making use of the `get_free_pages` function. The `zc_data` variable (`skb->zc_data`) points to the zero-copy data buffer (4096 Byte).

The fields of the new `sk_buff` data structure are listed in Table 5.1:

In order to use the zero-copy TCP-Socket, we have to assert `SO_ZERO_COPY` as socket-option by making use of the `setsockopt` function.

The size of a zero-copy package is 4096 Byte and the maximum transfer unit (MTU) of Gigabit Ethernet is 1500 Byte. We use the DMA-engine of the Gigabit Ethernet adapter in order to defragment packages. The defragmentation operation is implemented in `hamachi_rx` function (See Figure 5.1).

struct device	*dev	device that holds this sk_buff
unsigned char	*head	begin of buffer
unsigned char	*data	begin of data in buffer
unsigned char	*tail	end of data in buffer
unsigned char	len	length of data in buffer
unsigned char	*end	end of buffer
unsigned char	*zc_data	zero-copy data

Table 5.1: sk_buff data structure

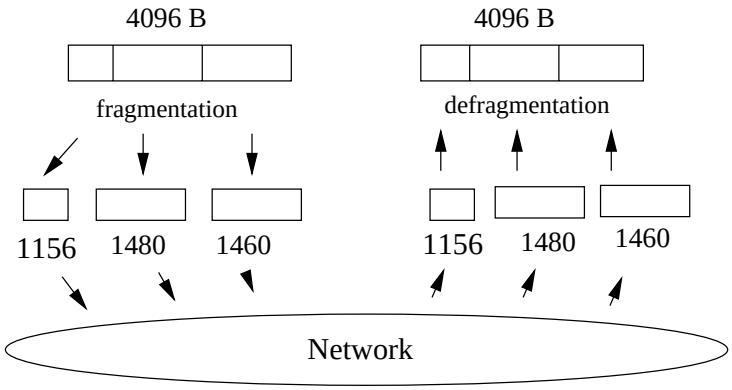

Figure 5.1: Fragmentation/Defragmentation operations

The Hamachi-Chip use the `get_free_pages` function in order to allocate enough space for the received buffers. This allocation takes place in `hamachi_init_ring` function. For each received packet, Hamachi allocates two physical pages. Since the received packet contains status info too, the second physical page will never be mapped into the user-space. Therefore, `sk_buff` data structure provided by the Hamachi-Chip to the kernel contains only the first allocated page.

When the kernel releases the `sk_buff` data structure, data pointed by the `zc_data` variable is deallocated by making use of the `free_pages` function. This operation takes place in `kfree_skmem` (net/core/skbuff.c)

The key to attaining efficiency in the implementation is to take advantage of info already loaded in the MMU tables. For a user buffer in an address space, the physical page behind it and its protection mode can be quickly retrieved from the MMU if the buffer is currently mapped. On the transmit side, when entering the *write* system call, the user buffer has to be mapped into the kernel address space.

Once the alignment requirement is met, very little change is needed in the networking code, except at the socket layer to replace *coyin* with *mapin+COW*, and *copyout* with *remap*. In the following we present the operations on both the sender side and the receiver side.

Sender side

The execution steps of the *write* system call are:

1. Find the physical page behind the current user buffer by making use of MMU tables (call the functions `pgd_offset`, `pmd_offset`, `pte_offset` and `pte_page`).
2. Change the user protection to read-only. Mark the user page as copy-on-write (by making use of `set_pte` function).
3. Map the physical page behind the current user buffer into the corresponding zero-copy data field of the `sk_buff` (`skb->zc_data`).

All these operations are implemented in the `tcp_do_sendmsg` function (net/ipv4/tcp.c).

The steps performed when the driver releases the `sk_buff` (physical page) are:

1. Unlock the page (by making use of the `clear_bit` function).
2. Do not unmap and deallocate the MMU resources.

All these operations are implemented in the `kfree_skmem` function (net/core/skbuff.c).

Receiver side

The execution steps of the *read* system call are:

1. Create a new Page Table Entry in the current process's Page Table and map the physical page address (found in `skb->zc_data` of `sk_buff` data structure) to it. We make use of `set_pte` and `mk_pte` functions.
2. Give the old user page back to the driver.

All these operations are implemented in the `tcp_recvmsg` function (net/ipv4/tcp.c).

To remap each page, the software must flush the obsolete entry from the local translation look-aside buffer (TLB). We make use of `__flush_tbl_one` function. Due to the fact that the header of the TCP/IP protocol has to have a fixed length, no options can be allowed anymore. This requires us to invalidate the TCP-Timestamp-Option by setting:

```
echo 0 > /proc/sys/net/ipv4/tcp_timestamps
```

To conclude, the existing prototype implementation of a zero-copy socket interface has the following **restrictions**:

1. The user buffers must be page align, and occupy an integral number of MMU pages.
2. The packets must be large enough to cover a whole page of data.
3. No TCP/IP options are allowed.

If the user buffer is smaller than a page size, the implementation preserves the copy semantics of traditional socket interface.

At the sender side, if the user buffer is larger than a page size, data has to be sent on the network in packets of 4096 Byte (as many as are possible). If data is not a multiple of page size, all packets of 4096 Byte will make use of the zero-copy layer and for the last packet (smaller than a page size) the copy semantics of traditional socket interface will be preserved.

Due to the defragmentation, data received by the kernel can be 4096 Byte (hold by a single page) or less in size. At the receiver side, if the receiving user buffer address is not page aligned and if the received data is not 4096 Byte, the copy semantics of traditional socket interface is used.

5.2 Integration of the existing zero-copy with MPICH

The existing prototype implementation of a "zero-copy" socket interface performs defragmentation of the received packages (at the device level). This operation leads to the restriction that *we only have point-to point communication.*

In order to determine MPICH to get use of the existing zero-copy socket implementation, there are some minimal changes to perform. First, we have to set the zero-copy socket-option (SO_ZERO_COPY). Then we have to set the length of the buffers written to the network to be equal with 4096 Byte. The modification has to be made in `net_send` function and consists of setting the value of the variable `trial_size` to the value of 4096 Byte (that is the page size for IBM/PC case).

One of the most important restriction of the existing zero-copy socket implementation is that the user buffers must be page align. MPICH can allocate page align buffers.

Regarding the MPICH implementation, as we have already mentioned, the **most important problem** is that at a lower level, data is received into a statically allocated structure. At the upper levels received data is copied into the corresponding field from posted or unexpected queues. This means that data is not directly received into the user buffer that is page align. Therefore, MPICH cannot make use of the zero-copy interface at the receiver side. We can see that for short messages (short protocol), data is sent with the control part of the message instead of being sent in a separate data part. This leads to a copy operation. Again this means that data is not directly sent from the user buffer that is page align. In case of long messages (eager protocol) the control message is sent first and after than the corresponding data. In this case data is sent without copies, directly from the user buffer that is page align. This means that at the sender side, for packages from 16384 Byte to 131072 Byte, MPICH makes use of the existing zero-copy socket interface.

The data way in MPICH is shown in Table 5.2. The information refers to long message (eager protocol) whose data size is 16384 Byte.

We can observe that the user buffer for sending and receiving data is page align but data is not directly received into the user buffer.

Initially there were some problems at the synchronization part between sender and receiver (device driver level). After the defragmentation operation is completed, the receiving ring buffers have to be refilled. After this operation the correct entry into the receiving queue

— send control data —	
send buf addr = 0xbffffad0	control packet size = 32 B
send buf addr = 0xbffffb5c	data size = 28 B
— send channel (data) —	
send buf addr = 0xbffffad0	control packet size =32 B
send buf addr = 0x80b4000	data size = 16384 B
— receive control data —	
recv buf addr = 0xbfffb9b4	read packet size = 32 B
recv buf addr = 0x80b8840	read data size = 28 B
COPY data into	
recv buf addr = 0xbfffbb04	
— receive channel (data) —	
recv buf addr = 0xbfffb984	read packet size = 32 B
recv buf addr = 0x80b88b0	read data size = 16384 B
COPY data into	
recv buf addr = 0x80b4000	

Table 5.2: The data way in MPICH

was lost.

Chapter 6

Fast Buffers (fbufs)

6.1 Overview

In this section we present the implementation of an efficient zero-copy framework for buffer management and exchange between application programs and the Linux kernel.

This "zero-copy" solution has the following main elements:

1. A buffer management scheme based on the *fast buffers (fbufs)* concept.
2. An extension of the basic fbufs concept to allow creation of fbufs out of memory-mapped files.
3. Extensions of the Linux Application Programming Interface (API) in the form of new calls to allocate and deallocate buffers. This API extension is implemented by a new library plus new system calls.
4. Extensions to the device driver kernel support routines to allow drivers to allocate and release fbufs.
5. Integration of buffer management with the virtual memory system for allocating and deallocating memory for the buffers.

We implemented a prototype that incorporates all the above elements as a loadable module. The modifications to the kernel were kept to the minimum.

The traditional UNIX I/O interfaces, such as `read` and `write` calls, are based on copy semantics, where data is copied between the kernel and user-defined buffers. On a `read` call, the user presents the kernel with a pre-allocated buffer, and the kernel must put the data read into it. On `write` call, the user is free to reuse the data buffer as soon as the call returns.

By passing an fbuf to the kernel on output, the application is guaranteed a saving in data-copying overheads which are implicitly performed with the traditional `write` system call. Similarly, by receiving data in an fbuf on input, the application is guaranteed a saving in data-copying overheads of the traditional `read` system call.

A Linux application may allocate fbufs, generate data in the fbufs, and transfer the fbufs to the kernel as part of output to a device. A second scenario is that the application obtains an fbuf containing data from a specified network device driver. The application may then process the data and either transfer the fbuf back to the kernel as part of output (to the same device driver) or simply deallocate it (see Figure 6.1).

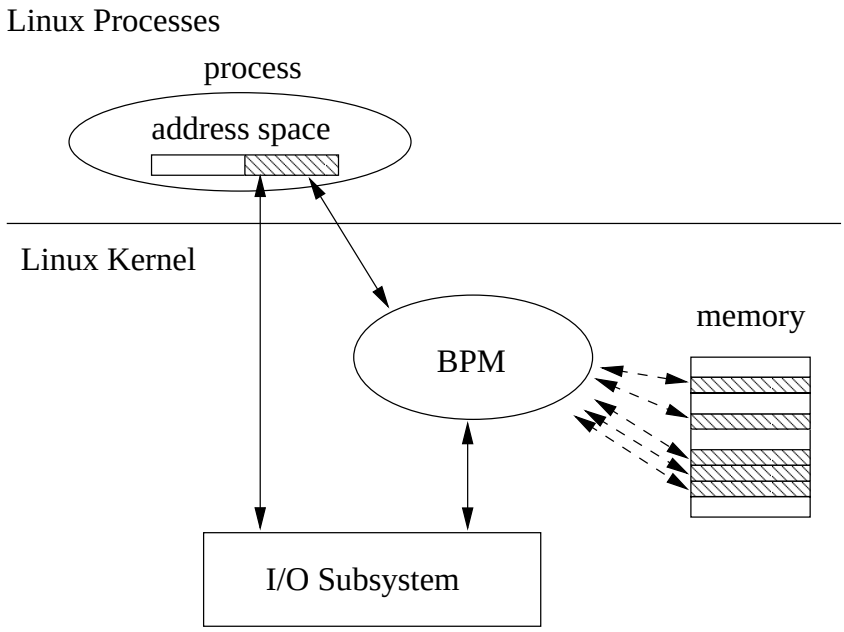

Figure 6.1: Fbuf allocation

The main implementation properties of fbufs are:

1. Each fbuf is either owned by an application program or by the kernel, or it is in the buffer pool manager. An fbuf can be transferred from an application program to the kernel or vice versa.

2. Each fbufs buffer pool is associated with an application program. An application program is associated with at most one buffer pool manager.

3. A network device driver can allocate fbufs and place incoming data directly into them. So, no copying between buffers is needed.

6.2 Application Programming Interface

The fbufs' application programming interface provides system calls and library routines to allocate and deallocate buffers.

The `uf_alloc` interface constructs a corresponding virtual address space into the process virtual memory (**addr*) and maps fbufs to it. The `uf_dealloc` function gives the fbufs

mapped to that virtual area back to the buffer pool. After this operation it releases the corresponding virtual address space (**addr*). The fbufs at the address **addr* are either explicitly allocated by the `uf_alloc` system call or implicitly as a result of `read` and `write` calls.

6.3 Programming using the New API

The new API required by our fbufs framework makes it necessary to use a slightly different programming interface from the standard I/O API. The changes are small and easy to implement.

First, a library `libfbufs.h` must be linked by the application that uses the framework. Then, the application must call the new `uf_alloc` and `uf_dealloc` system calls instead of `malloc` and `free` system calls. These new system calls will be used only for allocation/deallocation of data that will be sent to/received from the network.

One observation is that the user do not have to use `uf_alloc` function for data smaller than one page or not multiple of page size. Another remark is that data allocated with `uf_alloc` function has to be deallocated only by the `uf_dealloc` function. The user is not allowed to call `free` for data allocated with `uf_alloc` function or `uf_dealloc` for data allocated with `malloc` function.

6.4 Implementation

The main implementation parts are: a library (*libfbufs*), a buffer pool manager (BPM) and a system call component (see Figure 6.2).

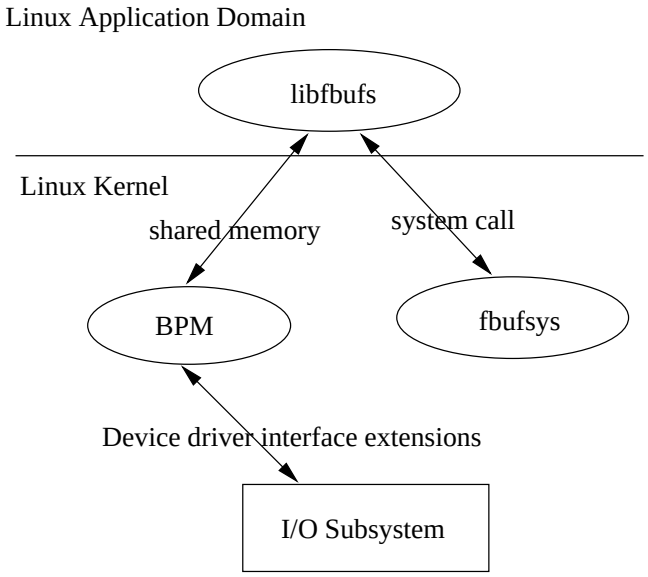

Figure 6.2: Linux fbuf implementation

6.4.1 libfbufs

The fbuf interface to Linux applications is provided by the *libfbufs* library that is running in the application domain. The application allocator/deallocator interfaces `uf_alloc` and `uf_dealloc` are implemented in this library and share data with the buffer pool manager. The `uf_alloc` and `uf_dealloc` interfaces are provided as traps to new system calls with the appropriate arguments set.

6.4.2 Buffer pool manager

Each application program that uses fbufs is associated with an instance of the buffer pool manager. The buffer pool manager is responsible for allocation of memory, tracking the allocation/deallocation of individual fbufs, managing mappings between user and kernel addresses.

6.4.2.1 Allocation/deallocation of memory

An fbufs consists of a single physical page. The internal representation of an fbufs in shown in Table 6.1.

```
struct fbuf {
 int context;                /* 0=kernel; 1=user */
 int usage;                  /* free page? 0=no; 1=yes */
 unsigned char *k_addr;      /* fbufs' kernel address */
 unsigned char *u_addr;      /* fbufs' user address if case */
 int device;
};
```

Table 6.1: The fbufs structure

A buffer pool manager (BPM) is composed by a number of `fbuf` data structures. Function `bpm_init` performs the BPM's initialization. The pages required by the BPM's `fbuf` data structures are allocated as pairs of two (2 pages that are consecutive in physical memory - suitable for DMA) as they are later requested by the Hamachi-Chip. These pages are allocated by making use of `__get_free_pages` kernel function and now are marked as free pages (are still not in use). At the BPM's initialization stage, these pages are in the kernel space (BPM is in the kernel space). The `bpm_init` function is called when the fbuf module is loaded. Memory allocated by BPM will be released by `bpm_close`. For this operation, the `free_pages` kernel function is called. `bpm_close` is called just before unloading fbuf module.

6.4.2.2 Locking/unlocking fbuf memory

One important observation is that the kernel will not have to handle any page fault when accessing the fbufs. To avoid the cost of locking the fbuf memory on each output operation and

unlocking on each input operation, the implementation locks the BPM's memory, such that no locking/unlocking of fbuf memory is needed. The locking operation takes place in `bpm_init` by making use of the `set_bit` function just after the BPM's physical pages are allocated. The unlocking operation takes place in `bpm_close` by means of `clear_bit` function just before the BPM's physical pages are released.

6.4.2.3 Managing mappings between user and kernel addresses

This prototype implementation of a zero-copy socket interface follows the principle of "User-kernel shared memory" zero-copy. It uses DMA to move data between the shared memory and the network interface. Because the user and kernel addresses of an fbuf will be different, the implementation re-maps buffers by editing the MMU table to give the appearance of data transfer. This means that all the buffers involved must align on page boundaries and occupy an integral number of MMU pages. Because of these restrictions, the fbuf implementation is based on the existing prototype implementation of a zero-copy socket interface. This means that we make use the new `sk_buff` data structure. This contains a new variable `zc_data` field pointing to the zero-copy data buffer (4096 Byte).

Due to the fact that a zero-copy package is 4096 Byte in size and the maximum transfer unit (MTU) of Gigabit Ethernet is 1500 Byte, we also have to get use of the DMA-engine of the Gigabit Ethernet adapter and to defragment packages. The defragmentation operation is implemented in `hamachi_rx`. Therefore, after defragmentation operation, zero-copy data given by the network device driver to the kernel (hold by `sk_buff` data structure) is 4096 Byte.

In order to allocate enough space for the received buffers, the Hamachi-Chip gets use of the `bpm_k_rq_two` BPM function. This function gives two physical pages (fbufs) from the BPM and marks them as being used in the kernel space. This allocation takes place in the `hamachi_init_ring` function. For each received packet, Hamachi requires two physical pages (suitable for DMA). Since the received packet contains status info too, the second physical page will never be mapped into the user-space. The second page is discarded in `hamachi_rx` function by calling `bpm_k_rel_sec` BPM function. This function gives a physical page (fbuf) back to the BPM and marks it as no longer being in use. Therefore, `sk_buff` data structure provided by the Hamachi-Chip contains only the first allocated page (one fbuf).

Like in the case of the existing prototype implementation of a zero-copy socket interface, the key to attaining efficiency in the implementation is to take advantage of info already loaded in the MMU tables. For a user buffer in an address space, the physical page behind it and its protection mode can be quickly retrieved from the MMU if the buffer is currently mapped. On the sender side, when entering the *write* system call, the user buffer has to be mapped into the kernel address space.

Once the alignment requirement is met, we have to replace *coyin* with *mapin+COW*, and *copyout* with *remap*. In the following, we present the operations on both the sender and the receiver side.

Sender side

The execution steps of the *write* system call are:

1. Find the physical page behind the current user buffer (the corresponding fbuf) by making use of MMU tables (call the functions `pgd_offset`, `pmd_offset`, `pte_offset` and `pte_page`).
2. Map this physical page (fbuf) into the corresponding zero-copy data field of the `sk_buff` (`skb->zc_data`).
3. Announce BPM that this fbuf is now into the kernel space (no longer into the user space) by means of `do_user_rel` function call.

All these operations are implemented in the `tcp_do_sendmsg` (net/ipv4/tcp.c)

When the kernel releases the `sk_buff` data structure, data pointed by the `zc_data` variable (one fbuf) is given back to the BPM by making use of `do_kernel_rel` function. This operation takes place in `kfree_skbmem` function (net/core/skbuff.c).

Receiver side

The execution steps of the *read* system call are:

1. Create a new Page Table Entry in the current process's Page Table and map the physical page address that in fact is a fbuf (found in `skb->zc_data` of `sk_buff` data structure) to it. We make use of `set_pte` and `mk_pte` functions.
2. Announce BPM that this fbuf is now into the user space by means of `do_kernel_to_user` function call.

All these operations are implemented in `tcp_recvmsg` (net/ipv4/tcp.c).

To remap each page, the software must flush the obsolete entry from the local translation look-aside buffer (TLB). We make use of `__flush_tbl_one` function.

As we have already mention, the application calls `uf_alloc` system call in order to make use of fbufs. The BPM function that gives the proper functionality to this system call is `do_fbufs_mmap`. It receives the required memory length, builds the corresponding user space for the required fbufs in the requesting process space, maps physical pages (fbufs) from the BPM to that virtual addresses and returns the corresponding user address (built here) to the requesting process. The BPM marks these fbufs as being in the user space and keeps the user address where these fbufs are mapped. Therefore, after the execution of the `uf_alloc` system call, there are fbufs mapped to that user virtual address. When data (fbufs) from the user buffer allocated by means of the `uf_alloc` system call is written to the network, the fbufs mapped there are passed to the TCP/IP stack by means of re-mapping operation. The result of attempting to access or modify that user buffer is undefined after the `write` function is called.

When packets are received by the network device driver, the defragmentation operation will take place. The `sk_buff` data structure given by the Hamachi-Chip to the kernel contains the received zero-copy data of 4096 Byte (one fbuf). After the `sk_buff` data structure is

passed to the kernel, in order to refill receiving buffers (fbufs) from the receiving queue, the Hamachi-Chip will call `bpm_k_rq_two` BPM function. Then, the received `sk_buff` data structure (a fbuf) is mapped to the corresponding user buffer address (allocated by means of the `uf_alloc` system call).

On the receiver side (at the moment of a `read` system call) two cases can be identified. In the first case, the `read` call follows a `write` call. That means before the `read` call there are no fbufs mapped to that user address. In the second case, the `read` call follows a `read` call. That means before a `read` call there are fbufs mapped to that user address and we have to give them back to the BPM before the new mapping operation takes place. This operation is performed by making use of `do_user_rel_exc` function. It announces the BPM that these fbufs are no longer mapped into the user space and are no longer in use.

As we have already presented, when an fbuf is passed from the user space to the kernel space and vice versa or when an fbuf is released by the kernel (`sk_buff` data structure is released by `kfree_skbmem` function) we have to announce the BPM about this modifications. This announcements are implemented as functions but their functionalities are defined only after the fbuf module is loaded. Our solutions is to initially implement all these functions as having dummy functionalities. The dummy functions are defined in net/ipv4/tcp.c file and declared in include/net/tcp.h. The dummy functionalities are set in `start_kernel` function (init/main.c). When the fbuf module will be loaded, we provide them with the proper functionalities and these are set in `init_module` function. Since we do not want to leave the system in an unstable state, the `cleanup_module` function will restore all these functions to their original states.

6.4.3 System calls

The real process to kernel communication mechanism, the one used by all processes, is the system call. When a process requests a service from the kernel (such as opening a file, forking to a new process, or requesting more memory), this is the mechanism at choice. In general, a process is not supposed to be able to access the kernel. It can't access kernel memory and it can't call kernel functions. The hardware of the CPU enforces this (that's the reason why it's called 'protected mode'). System calls are an exception to this general rule. What happens is that the process fills the registers with the appropriate values and then calls a special instruction which jumps to a previously defined location in the kernel (of course, that location is readable by user processes, it is not writable by them). Under Intel CPUs, this is done by means of interrupt. The hardware knows that once you jump to this location, you are no longer running in restricted user mode, but as the operating system kernel – and therefore you're allowed to do whatever you want. The location in the kernel a process can jump to is called *system_call*. The procedure at that location checks the system call number, which tells the kernel what service the process requested. Then, it looks at the table of system calls (`sys_call_table`) to see the address of the kernel function to call. Then it calls the function, and after it returns, does a few system checks and then return back to the process.

As we have already mentioned, the `uf_alloc` and `uf_dealloc` interfaces are provided as traps to new system calls with the appropriate arguments set. The implementation of a new system call is a tricky one. The main restriction is that a system call is part of the

kernel. That means that is has to have a corresponding entrance into the table of system calls (`sys_call_table`). The proper functionality of the `uf_alloc` system call (for example) is defined only after the fbuf module is loaded. Our solutions is to implement a dummy system call. When the fbuf module will be loaded, we will change the way this dummy system call works and we will provide its proper functionality. Due to the fact we don't want to leave the system in an unstable state, the `cleanup_module` function will restore the table to its original state.

In the following, we will present the implementation of a dummy system call. We will consider the case of `uf_alloc` function. `uf_dealloc` is implemented in a similar way.

The implementation steps of a new system call are:

1. Define the dummy functionality of the new system call (for `fbufs_ualloc` it is in /fbufs_ualloc.c).

2. Build a corresponding entrance for this file into Makefile.

3. Build a new system call number for the new system call (include/asm/unistd.h file).

4. Set the corresponding entry for the new system call in the table of system calls (in arch/i386/kernel/entry.S).

The fbufs module will change the dummy system call's functionality. The `init_module` function of the fbuf module replaces the appropriate location in the `sys_call_table` with the proper system call's functionality and keeps the original pointer (dummy functionality) in a variable (`original_all_call`). The `cleanup_module` function uses that variable to restore everything back to normal.

6.4.4 Impact of fbufs on the device drivers

We have to remember that a zero-copy package has 4096 Byte and the MTU of Gigabit Ethernet has 1500 Byte. Therefore, we have to perform the defragmentation operation in order to provide a `sk_buff` data structure holding a zero-copy data field of 4096 Byte. This operation is implemented in the `hamachi_rx` function. We will make use of the already modified hamachi device driver from the existing prototype implementation of a zero-copy interface. The changes required by fbufs are oriented around allocation/deallocation of fbufs.

When the network device driver module is loaded, it has to initialize its receiving queue. It will ask the BPM for a number of fbufs (by means of `bpm_k_rq_two` function). These fbufs are requested in order to keep the incoming packets.

On the other hand, when the network device driver module is unloaded, it has to free all fbufs hold by its receiving queue. This operation is performed by means of `bpm_k_rel_two` function call. This function gives all these fbufs back to the BPM.

6.4.5 Implementation notes and restrictions

The existing prototype implementation of a zero-copy socket interface has the following **restrictions**:

1. The user buffers must be page align, and occupy an integral number of MMU pages.
2. The packets must be large enough to cover a whole page of data.
3. No TCP/IP options are allowed.

We have to remember that the user is required not to use `uf_alloc` function for data smaller than one page in size or not a multiple of page size. If the user buffer is smaller than a page size or not a multiple of page size, the implementation preserves the copy semantics of traditional socket interface. In this case data was allocated by making use of `malloc` system call.

On the sender side, if the user buffer is greater than (and multiple of) a page size , data has to be sent on the network in packets of 4096 Byte (as many as are possible). In this case a packet is in fact a fbuf. Due to the defragmentation, data received by the kernel can be 4096 Byte (hold by a single page) or less in size. On the receiver side, if the receiving user buffer address is not page aligned and if the received data is not 4096 Byte, the copy semantics of traditional socket interface is used.

Because the header of TCP/IP protocol has to have a fixed length, no IP options are allowed. This requires us to invalidate the TCP-Timestamp-Option by setting:

```
echo 0 > /proc/sys/net/ipv4/tcp_timestamps
```

As we have already mentioned, the buffer pool manager is implemented as a loadable module. Once the fbuf module has been loaded it becomes a part of the kernel code. The kernel keeps a list of all of the kernel's resources in the kernel symbol table. When the fbus module is loaded, the kernel modifies the kernel symbol table, adding all the resources or symbols exported by this module (by making use of EXPORT_SYMBOL). When the module is unloaded, the kernel removes any symbol that the module exported into the kernel symbol table.

As we mentioned before, in order to allocate/deallocate fbufs for the received data, the network device driver will make use of some functions defined by the buffer pool manager. Therefore, one important remark is that the fbuf module has to be loaded/unloaded before performing the same operations for the network device driver.

Chapter 7

Performance Evaluation

7.1 Performance Evaluation of zero-copy implementations

A program called **ttcp** has been used for performance evaluation of zero-copy layers. This is a popular benchmark for measuring TCP throughput and its options are listed in Table 7.1.

In order to use the zero-copy socket layer, **ttcp** was slightly modified. Now, it allows a new option called `-z` that sets the `SO_ZERO_COPY` socket option:

```
-z      use zero copy sockets
        (sets SO_ZERO_COPY socket option)
```

The performance evaluation has been performed by making the following calls:

```
Sender:      ttcp -tszl 4096 -n x hostname
Receiver:    ttcp -rszl 4096
```

where “x” is the number of source buffers (ranging from 0 to 100000) written to network. The source buffer is 4096 Byte.

The TCP throughput with:

1. Traditional socket interface

   The TCP throughput without zero-copy layers is around 40 MB/s.

2. The existing zero-copy implementation

   The TCP throughput with the first version of zero-copy layer is approximately 60 MB/s (see Figure 7.1). The results have been obtained for various numbers of source buffers (ranging from 0 to 100000) written to the network. The source buffer is 4096 Byte.

3. Fbufs - The second zero-copy version

Usage:	ttcp -t [-options] host [< in]
	ttcp -r [-options > out]
Common options:	
-l	length of bufs read from or written to network (default 8192)
-u	use UDP instead of TCP
-p	port number to send to or listen at (default 5001)
-s	-t: source a pattern to network
	-r: sink (discard) all data from network
-A	align the start of buffers to this modulus (default 16384)
-O	start buffers at this offset from the modulus (default 0)
-v	verbose: print more statistics
-d	set SO_DEBUG socket option
-b	set socket buffer size (if supported)
-f X	format for rate: k,K = kilobit,byte; m,M = mega; g,G = giga
Options specific to -t:	
-n	number of source bufs written to network (default 2048)
-D	don't buffer TCP writes (sets TCP_NODELAY socket option)
Options specific to -r:	
-B	for -s, only output full blocks as specified by -l (for TAR)
-T	"touch": access each byte as it's read

Table 7.1: ttcp options

In order to use the fbufs, the new ttcp option (`-z`) that sets the `SO_ZERO_COPY` socket option is not enough. We also have to replace `malloc()` and `free()` system calls with the new `uf_alloc()` and `uf_dealloc()` systems call (see Section 6.3.).

The results from the Figure 7.1 has been obtained for various numbers of source buffers (ranging from 0 to 100000) written to the network. The source buffer was 4096 Byte. The TCP throughput of fbufs layer is approximately 70 MB/s but this layer is not yet stable and is not enough tested.

7.2 Performance Evaluation of MPICH over Gigabit Ethernet

MPICH is an example of the classical postal model. Both the sender and the receiver participate in the message exchange. The sender performs a send operation and the receiver issues a receive operation. These operations can be invoked in either order, blocking or non-blocking.

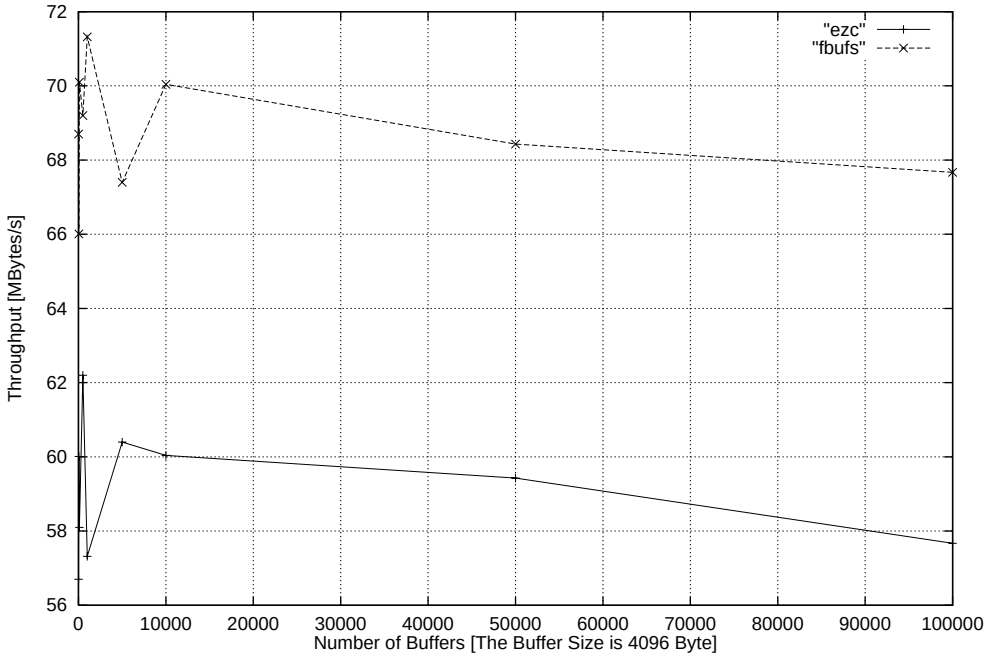

Figure 7.1: The TCP throughput with the existing zero-copy layer and with fbufs over Gigabit Ethernet

That is, messages can be sent at any time without waiting for the receiver, but this forces the system to buffer data until the receiver accepts it.

The typical amount of data transfered is usually too large to be stored in special purpose registers of the network interface. Therefore, buffering is done in software at a higher level of the message passing library and involves the memory system at the end points.

Latency and bandwidth depends as much on the efficiency of the TCP/IP implementation as on the network interface hardware and media.

The following results correspond to a point-to-point performance evaluation made on two Dell Optiplex GX1 (Pentium II, 350 MHz, 64 MByte RAM). For performance evaluation of MPICH over Gigabit Ethernet we made use of Packet Engines ”Hamachi” GNIC-II Network Interface Cards.

7.2.1 MPICH without zero-copy layer

In the following performance evaluations we made use of the 1-way function. The communication performance is measured using the benchmark (for MPI) provided by netlib (http://www.netlib.org/benckmark/index.html).

The performance of MPICH over Gigabit Ethernet matches the raw performance of 25 MByte/s for *blocking* sends and receive calls (see Fig 7.2). In this case, the maximum throughput value is achieved for a packet size of approximately 92681 Byte. For packets greater than

92681 Byte in size, the throughput is going to stabilized at the value of 22 MByte. This behavior is due to the fact that data larger than 65536 Byte is sent on the network in packets of at most 65536 Byte in size. The performance of message passing with the *non-blocking* send and receive calls has an exponential shape and the maximum throughput is 25 MByte/s for a packet size of 8192 Byte (see Fig 7.2). Its corresponding graph shows that the maximum packet size value for which we have communication is only of 8192 Byte (no throughput after this value). For packets larger then this value, due to data buffering, the message-passing transfer time increases and the communication is blocked (see Fig 7.3).

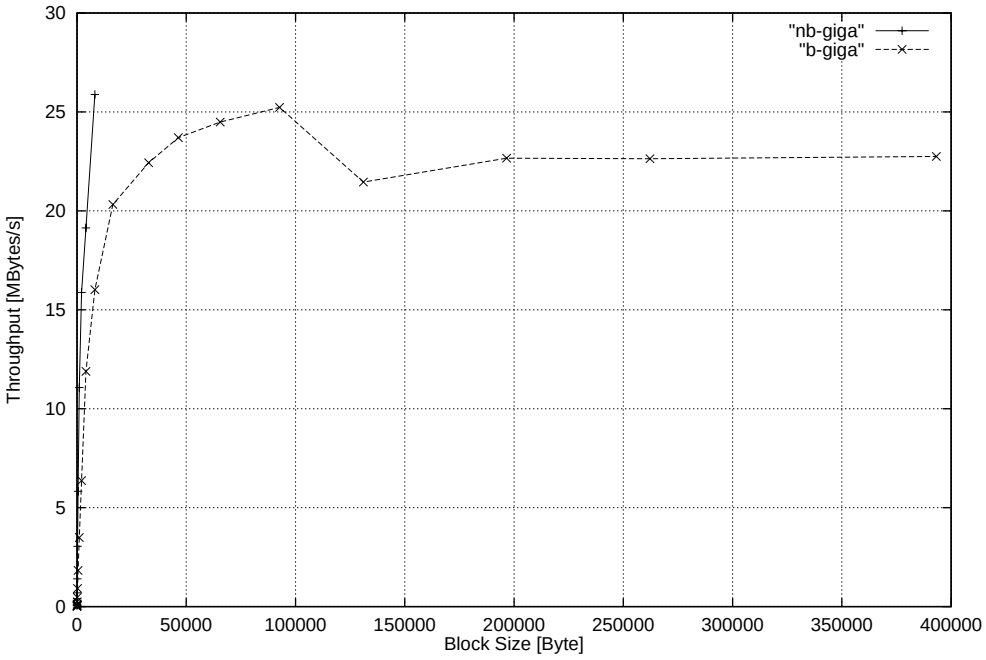

Figure 7.2: Measured throughput of MPICH without zero-copy layer over Gigabit Ethernet for non-blocking and blocking semantics

7.2.2 MPICH with zero-copy layer

Because of the integration problems described in section 5.2, the MPICH implementation cannot yet make use of the performance improvements offered by the zero-copy layers. The performance of the MPICH makeing use of the existing zero-copy layer is the same as without these layers (see Figure 7.4). In this case, the MPICH – zero-copy layer integration is simple (see Section 5.2).

The integration of the fbufs with applications is complex and the user has to take into consideration all the constraints described in section 6.3. Because of some instabilities we cannot yet provide the results for MPICH integrated with fbufs.

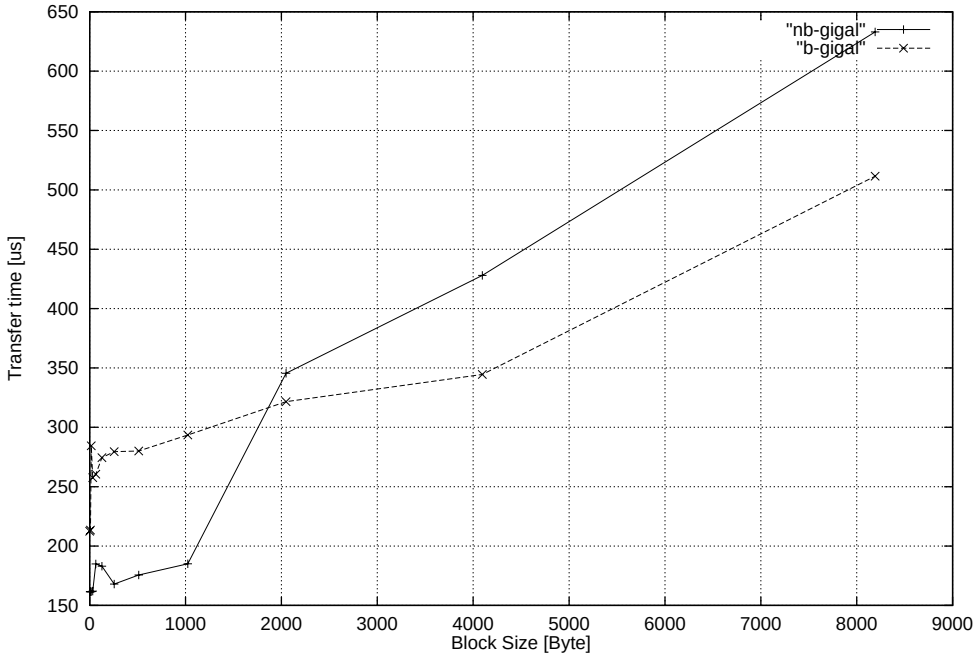

Figure 7.3: Mesage-passing transfer time in microseconds for MPICH transfers without zero-copy layer over Gigabit Ethernet for non-blocking and blocking semantics

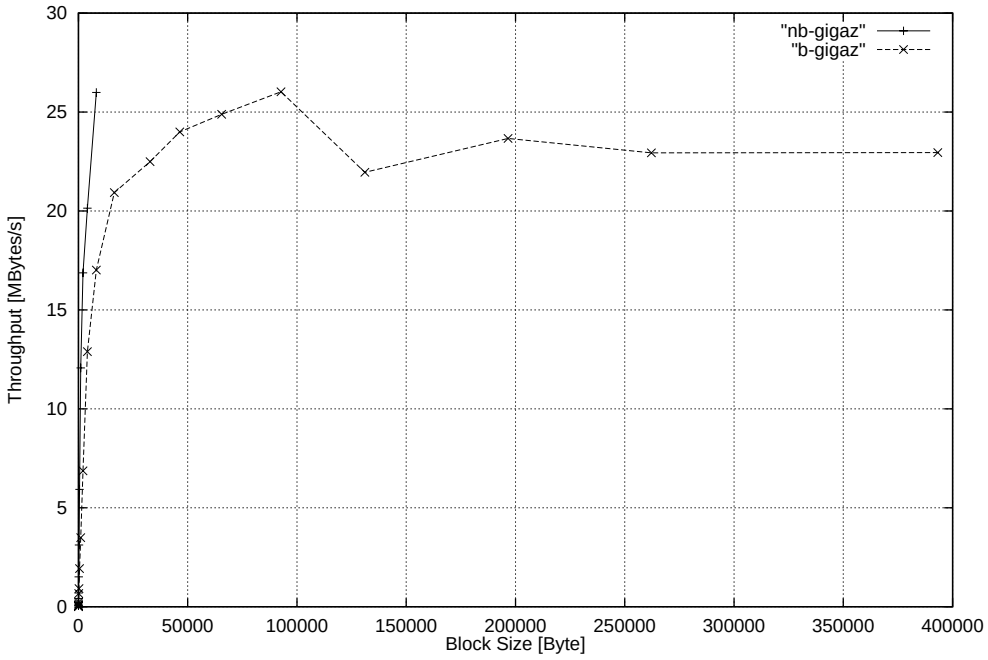

Figure 7.4: Measured throughput of MPICH with zero-copy layer over Gigabit Ethernet for non-blocking and blocking semantics

Appendix A

The code of the Buffer Pool Manager

```
/* bpm.c is the buffer pool manager that is responsable for allocation/
   deallocation of individual fbufs and satisfies kernel and user memory
   (fbufs) requests
*/

static int debug = 2;

#include <linux/config.h>
#ifdef MODULE
#ifdef MODVERSIONS
#include <linux/modversions.h>
#endif

#define EXPORT_SYMTAB
#include "linux/module.h"
#include "linux/version.h"
#else
#define MOD_INC_USE_COUNT
#define MOD_DEC_USE_COUNT
#endif

#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/string.h>
#include <linux/timer.h>
#include <linux/ptrace.h>
#include <linux/errno.h>
#include <linux/malloc.h>
#include <linux/interrupt.h>
#include <linux/mm.h>
#include <linux/slab.h>
#include <linux/shm.h>
```

```
#include <linux/mman.h>
#include <linux/pagemap.h>
#include <linux/swap.h>
#include <linux/swapctl.h>
#include <linux/smp_lock.h>
#include <linux/init.h>
#include <linux/file.h>

#include <linux/tcp.h>
#include <linux/skbuff.h>
#include <net/tcp.h>

#define vm_avl_empty    (struct vm_area_struct *) NULL
#include "/home/chihaia/zerocopy/linux-2.2.1/mm/mmap_avl.c"

#include <asm/processor.h> /* Processor type for cache alignment. */
#include <asm/uaccess.h>
#include <asm/pgtable.h>
#include <asm/page.h>

#include <linux/unistd.h>
#include <sys/syscall.h>

#define numar 200

/* The system call table (a table of functions). We just define
   this as external, and the kernel will fill it up for us when
   we are insmod'ed
*/
extern void *sys_call_table[];

/* Pointers to the original dummy system calls fbufs_ualloc() and
   fbufs_udealloc()
*/
asmlinkage unsigned long (*original_all_call)(unsigned long);
asmlinkage int (*original_deall_call)(unsigned long);


/* data in fbuf is 4098 long and page-allign */

struct fbuf {
int context; /* 0=kernel; 1=user */
int usage;       /* 0=no; 1=yes */
        unsigned char *k_addr;          /* kernel address */
        unsigned char *u_addr; /* user address if case */
int device;
};
```

```
struct fbuf m_list[numar];

static int bpm_init();
static void bpm_close();

static int bpm_test(unsigned char *u_addr);
static unsigned char* bpm_user_rq_one(unsigned char* u_addr);
static unsigned char* bpm_user_rq_two(unsigned char* u_addr);
int bpm_user_rel(unsigned char* u_addr);
int bpm_user_rel_exc(unsigned char* u_addr);

unsigned char* bpm_k_rq_two();
int bpm_k_rel_two(unsigned char* k_addr);
int bpm_k_rel_one(unsigned char* k_addr);
int bpm_k_rel_sec(unsigned char* k_addr);

int bpm_kernel_to_user(unsigned char* k_addr, unsigned char* u_addr);
unsigned long do_fbufs_mmap(unsigned long len);
int do_fbufs_munmap(unsigned long addr, unsigned long len);

extern int vm_enough_memory(long);
extern struct vm_area_struct * find_vma_prev
(struct mm_struct * , unsigned long, struct vm_area_struct **);


EXPORT_SYMBOL(bpm_user_rel);
EXPORT_SYMBOL(bpm_user_rel_exc);
EXPORT_SYMBOL(bpm_k_rq_two);
EXPORT_SYMBOL(bpm_k_rel_two);
EXPORT_SYMBOL(bpm_k_rel_one);
EXPORT_SYMBOL(bpm_k_rel_sec);
EXPORT_SYMBOL(bpm_kernel_to_user);
EXPORT_SYMBOL(do_fbufs_mmap);
EXPORT_SYMBOL(do_fbufs_munmap);


/* According to the fbuf concept the count flag of a fbuf
   (its corresponding physical page) is always 1: it is used
   by the appl or by the kernel or it is just kept in m_list
   The count flag is set to one in bpm_init() by
   __get_free_pages() and it is set to 0 in bpm_close() by
   free_page()
*/


/*  Initialization of the main vector keeping the pages used
```

```
    by the kernel and by the user. The pages are allocated as
    pairs of two (2 pages that are consecutive in physical
    memory - suitable for DMA) as requested by the hamachi chip
*/

static int bpm_init()
{
unsigned char * pages, * addr;
int i;

for (i=0; i < numar; i+=2)
{
pages = (unsigned char *)__get_free_pages(GFP_ATOMIC,1);
if (pages == NULL) {
printk(KERN_ERR "can't get free pages in init\n");
         return 0;
         } else {
        printk(KERN_DEBUG "pages=0x%x\n",(unsigned int)pages);
         }
         addr = pages;
         set_bit(PG_reserved,&mem_map[MAP_NR(addr)].flags);

         m_list[i].context = 0; /* used by kernel */
         m_list[i].usage = 0; /* not in use */
         m_list[i].k_addr = addr;
         m_list[i].u_addr = NULL;

         addr = (unsigned char *)((unsigned long)addr+PAGE_SIZE);

         set_bit(PG_reserved,&mem_map[MAP_NR(addr)].flags);
         m_list[i+1].context = 0;   /* used by kernel */
         m_list[i+1].usage = 0;     /* not in use */
         m_list[i+1].k_addr = addr;
         m_list[i+1].u_addr = NULL;
        }
        return 1;
}

/*  Before closing, bpm have to free the memory allocated by the
    initialization procedure
*/

static void bpm_close()
{
int i;

for (i=0; i < numar; i+=2)
```

```
{
printk(KERN_DEBUG "free_pages :0x%x\n", (int)m_list[i].k_addr);

clear_bit(PG_reserved,&mem_map[MAP_NR(m_list[i].k_addr)].flags);
clear_bit(PG_reserved,&mem_map[MAP_NR(m_list[i+1].k_addr)].flags);

free_pages((unsigned long)m_list[i].k_addr,1);

         m_list[i].context = 0; /* used by kernel */
         m_list[i].usage = 0; /* not in use */
         m_list[i].k_addr = NULL;
         m_list[i].u_addr = NULL;

         m_list[i+1].context = 0; /* used by kernel */
         m_list[i+1].usage = 0; /* not in use */
         m_list[i+1].k_addr = NULL;
         m_list[i+1].u_addr = NULL;
        }
}

/* an application requests fbufs - instead of malloc() for
   zc_data (only)!!! for mpich performance evaluation the
   modification takes place in mpi.c we will preserve malloc()
   for the other cases
*/

/* bpm_test() tests if at this user address there are mapped fbufs
   This test is called by do_fbufs_mmap() in order to unmap the
   user area
*/

static int bpm_test(unsigned char *u_addr)
{
int i;

for (i=0; i<numar; i++)
if (m_list[i].context && (m_list[i].u_addr == u_addr))
return 1;

return 0;
}


/* applic requests one physical page - one fbuf - we will return
   the fist free page from the first free pair of pages
*/
```

```
static unsigned char* bpm_user_rq_one(unsigned char *u_addr)
{
int i=0;
unsigned char *res=NULL;

while ((m_list[i].usage) && (i<numar))
i+=2;
if (i < numar ) {
m_list[i].context = 1; /* used by user */
m_list[i].usage = 1; /* in use */
/* user address is comming as parameter */
m_list[i].u_addr = u_addr;

res = m_list[i].k_addr;
}

return res;
}

/* Applic requests 2 physical pages - we will return the first
   free pair of pages
*/

static unsigned char* bpm_user_rq_two(unsigned char *u_addr)
{
int i=0;
unsigned char *res=NULL, * addr;

for(i=0; i<numar; i++)
{
if ((!m_list[i].usage) && (!m_list[i+1].usage) ) {
m_list[i].context = 1; /* used by user */
m_list[i].usage = 1; /* in use */
/* user address is comming as parameter */
addr = u_addr;
m_list[i].u_addr = addr;

m_list[i+1].context = 1; /* used by user */
m_list[i+1].usage = 1;  /* in use */
addr = (unsigned char *)((unsigned long)addr+PAGE_SIZE);
m_list[i+1].u_addr = addr;

res = m_list[i].k_addr;
i=numar;
} else {
  i+=2;
}
```

```
}

return res;
}

/* An application release a fbufs - in fct tcp_do_sendmsg()
   - this still will be used by the kernel for keeping
   skb->sz_data (remapping)
*/

int bpm_user_rel(unsigned char* u_addr)
{
int i;

for (i=0; i<numar; i++)
if (m_list[i].context && (m_list[i].u_addr == u_addr))
{
m_list[i].context = 0; /* back to the kernel */
m_list[i].u_addr = NULL;
return 1;
}
printk(KERN_ERR "error in user_rel\n");
return 0;
}

/* there can be exceptions in tcp_rcvmsg() when we have to free
   the old mappings and mark them as not in use also used in
   do_fbufs_munmap()
*/

int bpm_user_rel_exc(unsigned char* u_addr)
{
int i;

for (i=0; i<numar; i++)
if (m_list[i].context && (m_list[i].u_addr == u_addr))
{
m_list[i].context = 0; /* back to the kernel */
m_list[i].usage = 0; /* not in use */
m_list[i].u_addr = NULL;
return 1;
}
printk(KERN_ERR "error in user_rel_exc\n");
return 0;
}
```

```
/* the driver requests memory (always 2 consecutive pages
   - suitable for DMA) to hold zc_data
*/

unsigned char* bpm_k_rq_two()
{
int i=0;
unsigned char *res=NULL;

        for(i=0; i<numar; i++)
        {
         if ((!m_list[i].usage) && (!m_list[i+1].usage) ) {
                        m_list[i].usage = 1;
m_list[i+1].usage = 1;

res = m_list[i].k_addr;
i=numar;
} else {
i+=2;
}
}

return res;
}

/* Kernel release memory in kfree_skbmem(). Usually that page
   is the first one from the page pair but that can be the second
   one if the apllication have performed uf_alloc() for a buffer
   > PAGE_SIZE
*/

int bpm_k_rel_one(unsigned char* k_addr)
{
int i;

for (i=0; i<numar; i++)
if (m_list[i].k_addr == k_addr)
{
m_list[i].usage = 0; /* not in use */
return 1;
}
printk(KERN_ERR "error in k_rel_one\n");
return 0;
}

/* the driver release memory - in hamachi_close() */
```

```
int bpm_k_rel_two(unsigned char* k_addr)
{
int i;

for (i=0; i<numar; i+=2)
if (m_list[i].k_addr == k_addr)
{
m_list[i].usage = 0; /* not in use */
m_list[i+1].usage = 0; /* not in use */
return 1;
}
printk(KERN_ERR "error in k_rel_two\n");
return 0;
}

/* the driver release the second data page (status-info)
   - in hamachi_rx
*/

int bpm_k_rel_sec(unsigned char* k_addr)
{
int i;

for (i=1; i<numar; i+=2)
if (m_list[i].k_addr == k_addr)
{
m_list[i].usage = 0; /* not in use */
return 1;
}
printk(KERN_ERR "error in k_rel_sec\n");
return 0;
}

/* in tcp_recvmsg() the kernel is passing skb->zc_data to the user
*/

int bpm_kernel_to_user(unsigned char* k_addr, unsigned char* u_addr)
{
int i;

for (i=0; i<numar; i+=2)
if (m_list[i].k_addr == k_addr)
{
m_list[i].context = 1;  /* used by user */
                 /* user address is comming as parameter */
                        m_list[i].u_addr = u_addr;
return 1;
```

```
}
printk(KERN_ERR "error in kernel_to_user\n");
return 0;
}



/* Remove one vm structure from the inode's i_mmap ring. */
static inline void remove_shared_vm_struct(struct vm_area_struct *vma)
{
        struct file * file = vma->vm_file;

        if (file) {
         if (vma->vm_flags & VM_DENYWRITE)
                 file->f_dentry->d_inode->i_writecount++;
                if(vma->vm_next_share)
                        vma->vm_next_share->vm_pprev_share =\
 vma->vm_pprev_share;
                *vma->vm_pprev_share = vma->vm_next_share;
        }
}



/* Combine the mmap "prot" and "flags" argument into one "vm_flags" used
 * internally. Essentially, translate the "PROT_xxx" and "MAP_xxx" bits
 * into "VM_xxx".
 */
static inline unsigned long vm_flags
(unsigned long prot, unsigned long flags)
{
#define _trans(x,bit1,bit2) \
((bit1==bit2)?(x&bit1):(x&bit1)?bit2:0)

unsigned long prot_bits, flag_bits;
        prot_bits =
         _trans(prot, PROT_READ, VM_READ) |
                _trans(prot, PROT_WRITE, VM_WRITE) |
                _trans(prot, PROT_EXEC, VM_EXEC);
        flag_bits =
                _trans(flags, MAP_GROWSDOWN, VM_GROWSDOWN) |
                _trans(flags, MAP_DENYWRITE, VM_DENYWRITE) |
                _trans(flags, MAP_EXECUTABLE, VM_EXECUTABLE);
        return prot_bits | flag_bits;
#undef _trans
}
```

```
/* Normal function to fix up a mapping
 * This function is the default for when an area has no specific
 * function.  This may be used as part of a more specific routine.
 * This function works out what part of an area is affected and
 * adjusts the mapping information.  Since the actual page
 * manipulation is done in do_mmap(), none need be done here,
 * though it would probably be more appropriate.
 *
 * By the time this function is called, the area struct has been
 * removed from the process mapping list, so it needs to be
 * reinserted if necessary.
 *
 * The 4 main cases are:
 *    Unmapping the whole area
 *    Unmapping from the start of the segment to a point in it
 *    Unmapping from an intermediate point to the end
 *    Unmapping between to intermediate points, making a hole.
 *
 * Case 4 involves the creation of 2 new areas, for each side of
 * the hole.  If possible, we reuse the existing area rather than
 * allocate a new one, and the return indicates whether the old
 * area was reused.
 */

static int unmap_fixup(struct vm_area_struct *area,
unsigned long addr, size_t len,
struct vm_area_struct **extra)
{
struct vm_area_struct *mpnt;
unsigned long end = addr + len;

area->vm_mm->total_vm -= len >> PAGE_SHIFT;
if (area->vm_flags & VM_LOCKED)
area->vm_mm->locked_vm -= len >> PAGE_SHIFT;

/* Unmapping the whole area. */
if (addr == area->vm_start && end == area->vm_end) {
if (area->vm_ops && area->vm_ops->close)
area->vm_ops->close(area);
if (area->vm_file)
fput(area->vm_file);
return 0;
}

/* Work out to one of the ends. */
if (end == area->vm_end)
```

```
area->vm_end = addr;
else if (addr == area->vm_start) {
area->vm_offset += (end - area->vm_start);
area->vm_start = end;
} else {
/* Unmapping a hole:
   area->vm_start < addr <= end < area->vm_end */
/* Add end mapping -- leave beginning for below */
mpnt = *extra;
*extra = NULL;

mpnt->vm_mm = area->vm_mm;
mpnt->vm_start = end;
mpnt->vm_end = area->vm_end;
mpnt->vm_page_prot = area->vm_page_prot;
mpnt->vm_flags = area->vm_flags;
mpnt->vm_ops = area->vm_ops;
mpnt->vm_offset = area->vm_offset + (end - area->vm_start);
mpnt->vm_file = area->vm_file;
mpnt->vm_pte = area->vm_pte;
if (mpnt->vm_file)
mpnt->vm_file->f_count++;
if (mpnt->vm_ops && mpnt->vm_ops->open)
mpnt->vm_ops->open(mpnt);
area->vm_end = addr; /* Truncate area */
insert_vm_struct(current->mm, mpnt);
}

insert_vm_struct(current->mm, area);
return 1;
}

/*
 * Try to free as many page directory entries as we can,
 * without having to work very hard at actually scanning
 * the page tables themselves.
 *
 * Right now we try to free page tables if we have a nice
 * PGDIR-aligned area that got free'd up. We could be more
 * granular if we want to, but this is fast and simple,
 * and covers the bad cases.
 *
 * "prev", if it exists, points to a vma before the one
 * we just free'd - but there's no telling how much before.
 */

static void free_pgtables(struct mm_struct * mm,
```

```
struct vm_area_struct *prev,
unsigned long start, unsigned long end)
{
unsigned long first = start & PGDIR_MASK;
unsigned long last = (end + PGDIR_SIZE - 1) & PGDIR_MASK;

if (!prev) {
prev = mm->mmap;
if (!prev)
goto no_mmaps;
if (prev->vm_end > start) {
if (last > prev->vm_end)
last = prev->vm_end;
goto no_mmaps;
}
}
for (;;) {
struct vm_area_struct *next = prev->vm_next;

if (next) {
if (next->vm_start < start) {
prev = next;
continue;
}
if (last > next->vm_start)
last = next->vm_start;
}
if (prev->vm_end > first)
first = prev->vm_end + PGDIR_SIZE - 1;
break;
}
no_mmaps:
first = first >> PGDIR_SHIFT;
last = last >> PGDIR_SHIFT;
if (last > first)
clear_page_tables(mm, first, last-first);
}



/* fct do_fbufs_mmap() is called by the application by using
   uf_alloc() system call and is used instead of malloc()
   It receives the required memory length, builds the corresponding
   user space for the required fbufs in the requesting process
   space, maps physical pages from the bpm to that virtual
   addresses and returns the corresponding user address (built)
   to the requesting process
```

```
*/

unsigned long do_fbufs_mmap(unsigned long len)
{
        struct mm_struct * mm = current->mm;
        struct vm_area_struct * vma;
        unsigned char * phy_addr, *ptrcaddr;
        unsigned long length, curr_addr;

        unsigned long addr=0;
unsigned long prot=PROT_WRITE;
        unsigned long flags=MAP_SHARED;
        unsigned long off=0;

        pgd_t *pgd;
        pmd_t *pmd;
        pte_t *pte;
        int error;

printk("se intra cu len %d \n",len);
        if ((len = PAGE_ALIGN(len)) == 0)
                return addr;

        if (len > TASK_SIZE || addr > TASK_SIZE-len)
                return -EINVAL;

        /* offset overflow? */
        if (off + len < off)
                return -EINVAL;

        /* Too many mappings? */
        if (mm->map_count > MAX_MAP_COUNT)
                return -ENOMEM;

        /* mlock MCL_FUTURE? */
        if (mm->def_flags & VM_LOCKED) {
                unsigned long locked = mm->locked_vm << PAGE_SHIFT;
                locked += len;
                if (locked > current->rlim[RLIMIT_MEMLOCK].rlim_cur)
                        return -EAGAIN;
        }

        /* Obtain the address to map to. We verify (or select)
           it and ensure
           that it represents a valid section of the address space.
           flags=MAP_SHARED
         */
```

```
        addr = get_unmapped_area(addr, len);
         if (!addr)
                        return -ENOMEM;

        vma = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
        if (!vma)
                return -ENOMEM;

        vma->vm_mm = mm;
        vma->vm_start = addr;
        vma->vm_end = addr + len;
        vma->vm_flags = vm_flags((unsigned long)prot,\
(unsigned long)flags) | mm->def_flags;

        vma->vm_flags |= VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;
        vma->vm_page_prot = protection_map[vma->vm_flags & 0x0f];
        vma->vm_ops = NULL;
        vma->vm_offset = off;
        vma->vm_file = NULL;
        vma->vm_pte = 0;

        /* Clear old maps */
        error = -ENOMEM;
        ptrcaddr = (unsigned char *)vma->vm_start;
        if (bpm_test(ptrcaddr)) {
         printk("problema cu bpm_test \n");
         if (do_fbufs_munmap(addr, len))
                 goto free_vma;
        } else {
         printk("bpm_teste ok \n");
         if (do_munmap(addr, len))
         goto free_vma;
        }

        /* Check against address space limit. */
        if ((mm->total_vm << PAGE_SHIFT) + len
            > current->rlim[RLIMIT_AS].rlim_cur)
                goto free_vma;

        /* Private writable mapping? Check memory availability.. */
        if ((vma->vm_flags & (VM_SHARED | VM_WRITE)) == VM_WRITE &&
            !(flags & MAP_NORESERVE)                             &&
            !vm_enough_memory(len >> PAGE_SHIFT))
                goto free_vma;

/* Assign physical pages to the vma. We will check if there
```

```
   can be assigned 2 consecutive pages (as allocated in bpm_init)
   or just one page
*/

length=len;
curr_addr=vma->vm_start;

while (length > 0) {
if (length>PAGE_SIZE)
{
ptrcaddr=(unsigned char *)curr_addr;
         phy_addr=bpm_user_rq_two(ptrcaddr);
         if (phy_addr == NULL) {
         error = -ENOMEM;
                 goto unmap_and_free_vma;
         }
pgd=pgd_offset(current->mm, curr_addr);
pmd=pmd_offset(pgd, curr_addr);
pte=pte_offset(pmd, curr_addr);

set_pte(pte, mk_pte(phy_addr, PAGE_SHARED));
__flush_tlb_one(curr_addr);

curr_addr+=PAGE_SIZE;
length-=PAGE_SIZE;
} else {
phy_addr = bpm_user_rq_one(ptrcaddr);
}

if (phy_addr == NULL) {
         error = -ENOMEM;
                 goto unmap_and_free_vma;
         }
pgd=pgd_offset(current->mm, curr_addr);
pmd=pmd_offset(pgd, curr_addr);
pte=pte_offset(pmd, curr_addr);

set_pte(pte, mk_pte(phy_addr, PAGE_SHARED));
__flush_tlb_one(curr_addr);


if (length>PAGE_SIZE) {
length-=PAGE_SIZE;
curr_addr+=PAGE_SIZE;
} else {
length=0;
}
```

```
}

        /*
         * merge_segments may merge our vma, so we can't refer to it
         * after the call.  Save the values we need now ...
         */
        flags = vma->vm_flags;
        addr = vma->vm_start; /* can addr have changed?? */
        insert_vm_struct(mm, vma);
        merge_segments(mm, vma->vm_start, vma->vm_end);

        mm->total_vm += len >> PAGE_SHIFT;
        if (flags & VM_LOCKED) {
                mm->locked_vm += len >> PAGE_SHIFT;
                make_pages_present(addr, addr + len);
        }
        return addr;

unmap_and_free_vma:
        /* Undo any partial mapping done by a device driver. */
        flush_cache_range(mm, vma->vm_start, vma->vm_end);
        zap_page_range(mm, vma->vm_start,
               vma->vm_end - vma->vm_start);
        flush_tlb_range(mm, vma->vm_start, vma->vm_end);
free_vma:
        kmem_cache_free(vm_area_cachep, vma);
        return error;
}

/* fct do_fbufs_munmap() is called by the application by
   using uf_dealloc() system call and is used instead of
   free() It receives the the user address and the corresponding
   length, clears it and returns its corresponding fbufs
   back to the bpm
*/


int do_fbufs_munmap(unsigned long addr, unsigned long len)
{
struct mm_struct * mm;
struct vm_area_struct *mpnt, *prev, **npp, *free, *extra;

printk("addr e %x \n",addr);
printk("len e %d \n",len);

if ((addr & ~PAGE_MASK) || addr > TASK_SIZE || \
            len > TASK_SIZE-addr)
```

```
return -EINVAL;

if ((len = PAGE_ALIGN(len)) == 0)
return 0;

/* Check if this memory area is ok - put it on the temporary
 * list if so..  The checks here are pretty simple --
 * every area affected in some way (by any overlap) is put
 * on the list.  If nothing is put on, nothing is affected.
 */
mm = current->mm;
mpnt = find_vma_prev(mm, addr, &prev);
if (!mpnt)
return 0;
/* we have  addr < mpnt->vm_end  */

if (mpnt->vm_start >= addr+len)
return 0;

/* If we'll make "hole", check the vm areas limit */
if ((mpnt->vm_start < addr && mpnt->vm_end > addr+len)
    && mm->map_count >= MAX_MAP_COUNT)
return -ENOMEM;

/*
 * We may need one additional vma to fix up the mappings ...
 * and this is the last chance for an easy error exit.
 */
extra = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
if (!extra)
return -ENOMEM;

npp = (prev ? &prev->vm_next : &mm->mmap);
free = NULL;
for ( ; mpnt && mpnt->vm_start < addr+len; mpnt = *npp) {
*npp = mpnt->vm_next;
mpnt->vm_next = free;
free = mpnt;
if (mm->mmap_avl)
avl_remove(mpnt, &mm->mmap_avl);
}

/* Ok - we have the memory areas we should free on the 'free' list,
 * so release them, and unmap the page range..
 * If the one of the segments is only being partially unmapped,
 * it will put new vm_area_struct(s) into the address space.
 */
```

```
while ((mpnt = free) != NULL) {
unsigned long st, end, size, length, curr_addr;
unsigned char * ptrcaddr;

free = free->vm_next;

st = addr < mpnt->vm_start ? mpnt->vm_start : addr;
end = addr+len;
end = end > mpnt->vm_end ? mpnt->vm_end : end;
size = end - st;

/* return the fbuf back to the bpm */

length=size;
curr_addr=st;

while (length > 0)
{
ptrcaddr=(unsigned char*)curr_addr;
printk("ptrcaddr e %x \n",ptrcaddr);
         bpm_user_rel_exc(ptrcaddr);
if (length>PAGE_SIZE) {
length-=PAGE_SIZE;
curr_addr+=PAGE_SIZE;

} else {
length=0;
}
}

remove_shared_vm_struct(mpnt);
mm->map_count--;

flush_cache_range(mm, st, end);
zap_page_range(mm, st, size);
flush_tlb_range(mm, st, end);

/*
 * Fix the mapping, and free the old area if it wasn't reused.
 */
if (!unmap_fixup(mpnt, st, size, &extra))
kmem_cache_free(vm_area_cachep, mpnt);
}

/* Release the extra vma struct if it wasn't used */
if (extra)
kmem_cache_free(vm_area_cachep, extra);
```

```
free_pgtables(mm, prev, addr, addr+len);

mm->mmap_cache = NULL; /* Kill the cache. */
return 0;
}



/* proper_sys_call() is the function we'll replace the initial dummy
   sys_fbufs_ualloc() with
*/
asmlinkage unsigned long proper_sys_call(unsigned long length)
{
return do_fbufs_mmap((unsigned long) length);
}


/* proper_sys_all_call() is the function we'll replace the ini-
tial dummy
   sys_fbufs_ualloc() with
*/
asmlinkage unsigned long proper_sys_all_call(unsigned long length)
{
return do_fbufs_mmap((unsigned long) length);
}

/* proper_sys_deall_call() is the function we'll replace the
   initial dummy sys_fbufs_udealloc() with
*/
asmlinkage int proper_sys_deall_call
                  (unsigned long addr, unsigned long length)
{
return do_fbufs_munmap((unsigned long) addr, \
               (unsigned long) length);
}



#ifdef MODULE
int init_module(void)
{
int i;

/* Keep a pointer to the dummy fcts fbufs_ualloc() and fbufs_udealloc()
   in original_all_call and in original_deall_call, and then replace
```

```
   the dummy system calls in the system call table with
   proper_sys_all_call and with proper_sys_deall_call
*/

original_all_call = sys_call_table[__NR_fbufs_ualloc];
sys_call_table[__NR_fbufs_ualloc] = proper_sys_all_call;

original_deall_call = sys_call_table[__NR_fbufs_udealloc];
sys_call_table[__NR_fbufs_udealloc] = proper_sys_deall_call;

i = bpm_init();
if (!i) return 1;

do_kernel_to_user = &bpm_kernel_to_user;
do_user_rel = &bpm_user_rel;
do_user_rel_exc = &bpm_user_rel_exc;
do_kernel_rel = &bpm_k_rel_one;

return 0;
}

void cleanup_module(void)
{

/* Return the system calls sys_fbufs_ualloc and fbufs_udealloc back to
   their dummy functionalities
*/

if (sys_call_table[__NR_fbufs_ualloc] != proper_sys_all_call) {
printk("It seems that there are 2 bpm modules\n");
}
sys_call_table[__NR_fbufs_ualloc] = original_all_call;

if (sys_call_table[__NR_fbufs_udealloc] != proper_sys_deall_call) {
printk("It seems that there are 2 bpm modules\n");
}
sys_call_table[__NR_fbufs_udealloc] = original_deall_call;

        bpm_close();
        do_kernel_to_user = &dummy_kernel_to_user;
        do_user_rel = &dummy_user_rel;
        do_user_rel_exc = &dummy_user_rel_exc;
        do_kernel_rel = &dummy_k_rel_one;
}
#endif
```

Bibliography

[1] MPI Forum. MPI 2 Standard. 1997. http://www.mpi-forum.org.

[2] N.Doss W. Gropp, E. Lusk and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. In *Parallel Computing*, pages 789–828, sep 1996.

[3] W. Gropp and E. Lusk. *User's Guide for mpich, a portable implementation of MPI*. Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

[4] H. K. Jerry Chu. Zero-Copy TCP in Solaris. In *Proceedings of the USENIX 1996 Annual Technical Conference*, pages 253–264. USENIX Association, 1996.

[5] Peter Druschel and Larry Peterson. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. *Operating Systems Review*, December 1993.

[6] Moti N. Thadani and Yousef A. Khalidi. An Efficient Zero-Copy I/O Framework for UNIX. *Sun Microsystems*, May 1995.

[7] Michel Muller. *Zero Copy TCP/IP mit Gigabit Ethernet*. 1999. ETH Diploma Thesis.

[8] Packet Engines, Inc. G-NIC II Hamachi Engineering Design Specification. http://www.packetengines.com/. Packet Engines Homepage.

[9] Alessandro Rubini. *Linux Device Drivers*. O' Reilly & Associates, first edition, 1998. ISBN 1-56592-292-1.

[10] David A. Rusling. *The Linux Kernel*. The Linux Documentation Project, January 1999. Version 0.8-3.

[11] Michael Beck, Harald Bohme, Mirko Dziadzka, Robert Magnus, and Dirk Verworner. *Linux-Kernel-Programmierung*. Addison-Wesley, fourth edition, 1997. ISBN 3-8273-1144-6.

[12] Thomas M. Stricker, Christian Kurmann, Michela Taufer, and Felix Rauch. CoPs – Clusters of PCs. Project overview: http://www.cs.inf.ethz.ch/CoPs/index.html.