Mach-Based Channel Library

Rajit Manohar
Advisor: K. Mani Chandy

Department of Computer Science
California Institute of Technology
Pasadena, CA 91125.

July 7, 1994

In partial Fulfillment of the Requirements
for the degree of
Master of Science

Acknowledgments

I foremost want to thank my research advisor Prof. K. Mani Chandy for his guidance and support.

I would like to thank the members of my research group—Peter Carlin, Peter Hofstee, Svetlana Kryukova, K. Rustan M. Leino, Berna Massingill, Adam Rifkin, Paul Sivilotti, John Thornley—for comments, constructive criticism, and invaluable discussions.

This research was supported in part by Air Force Office of Scientific Research grant number 91-0070.

Contents

Chapter 1

Introduction

1.1 Motivation

We think of a parallel computation to be one that involves a number of tasks cooperating in some fashion. The manner in which the tasks cooperate becomes an important issue. There are two basic paradigms that describe cooperation between tasks:

1. Use of Shared Variables. In this technique the cooperating tasks share certain data structures, which are used to exchange information.

2. Message Passing. The message passing paradigm constrains the parallel computation in two ways: (i) tasks do not share any data; (ii) the only way that tasks can exchange information is by sending messages to each other. Sometimes, the first constraint is relaxed, resulting in a computation that uses both shared variables and messages.

Mach, an operating system developed at Carnegie Mellon University [1], provides support for both these paradigms. However, the flexibility provided by the operating system makes it difficult to reason about programs. The support provided for message passing focuses the attention of a user on the details of the communication layer provided by the operating system.

The goal of this thesis is to write a *small* message-passing library—using the support provided by Mach—for parallel computation. This library should be easy to use, language independent, and robust. It should focus attention on the message passing abstraction itself, rather than on the details of the underlying operating system. By imposing restrictions on the functions that are available, the library should simplify arguments involving program correctness.

The model chosen for the library is the one provided by Mach itself, in which tasks can communicate with one another by sending messages. Communication streams known as *channels* are provided that allow the user to think about the logical connections between cooperating tasks. Since the concept of channels is central to the library, we call it a channel library.

Mach provides kernel support for message passing. Since all Mach services are accessed through the message passing mechanism, the message passing mechanism provided is robust. As will be seen in chapter 2, the operating system enforces a system of access rights, which prevent tasks from exchanging messages unless they have permission to do so. In addition, the message passing mechanism provided is reliable, even across machines. Finally, the operating system provides kernel support for fair threads. We chose the Mach operating system as our base because of these features.

The functions provided by the channel library can be grouped into four basic categories; functions for:

1. Initialization.
2. Creating channels.
3. Sending and receiving messages.
4. Creating new tasks or threads.

1.2 Overview

Chapter 2 provides a brief introduction to the relevant concepts of the Mach operating system. Chapter 3 describes the interface provided by the channel library, by specifying each function provided by the library. In chapter 4, the implementation of the library is discussed in detail. Chapter 5 shows some of the major properties and invariants maintained by the library, and indicates how a rigorous proof of the library implementation could be constructed. Chapter 6 provides a few example programs that illustrate how the library is to be used.

Chapter 2

Introduction to Mach

2.1 Overview

The Mach operating system was developed at Carnegie Mellon University (CMU). The goal of the project was to write an operating system with a small kernel that would provide a few powerful primitives that a programmer could use to construct complex programs. With this in mind, the Mach kernel has been written to provide three basic services:

- Scheduling.
- Inter-process communication.
- Virtual memory management.

All other features are built on top of these three basic services. As a result, a number of services that are part of the kernel of traditional operating systems, such as UNIX [2], are user-level applications that run on top of the Mach kernel.

The Mach operating system is compatible with 4.3BSD UNIX, and most BSD programs can be executed under the Mach operating system without modification after recompilation.

Our main concerns when implementing the channel library on top of Mach are the following:

- Process creation.
- Thread creation.
- Communication.

We will therefore restrict our attention in this chapter to describing the scheduling and inter-process communication (IPC) aspects of the Mach kernel.

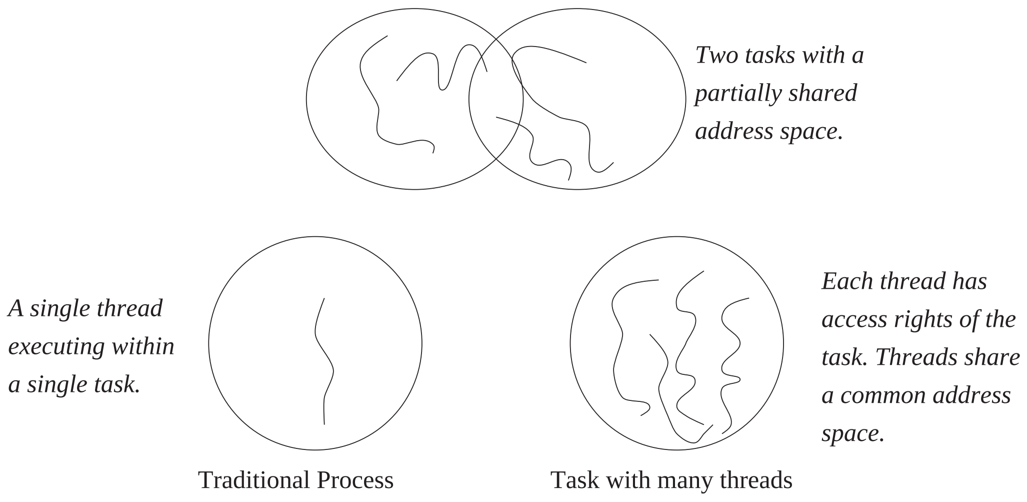
Two tasks with a partially shared address space.
A single thread executing within a single task.
Each thread has access rights of the task. Threads share a common address space.
Traditional Process
Task with many threads

Figure 2.1: Tasks and Threads

2.2 Threads and Tasks

Under the Mach operating system, the concept of a process has been divided into two parts: the *task* and the *thread*.

A task is an environment in which execution of a program occurs. The task consists of a virtual address space and access rights which give the task access to certain services and resources provided by the system. The virtual address space of a task cannot be accessed by another task unless explicit permission to do so is granted by the task.

A thread is the unit of execution. Threads execute within a task, which provides an environment for execution. Threads within a task share access rights of the task and the virtual address space of the task. Many threads can execute within a single task, and they all share the same address space as shown in fig. 2.1.

Thus, a traditional process consists of a single thread executing within a single task.

2.2.1 Scheduling

Mach provides a flexible thread scheduling method. Each thread has associated with it a *priority* and a *policy*. The policy indicates the scheduling method to be used for the thread. We will discuss the *timesharing* policy only, since it is the default, and the only policy we will use.

The timesharing scheduling policy ensures that execution of threads is weakly fair: every thread executes eventually. To ensure this, each thread is assigned a priority which decreases with the execution time of the thread. As a result, a thread with a high priority can never starve a low priority thread, thus ensuring that execution is weakly fair.

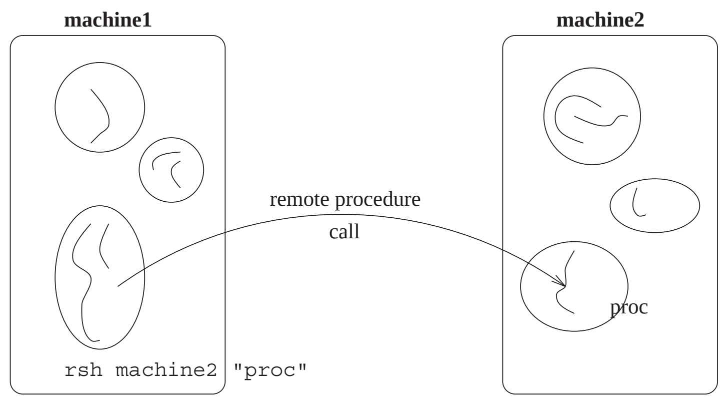

Figure 2.2: Remote Shell Program

2.2.2 Creating Tasks and Threads

A thread is defined by specifying the state of the thread, i.e., the state of the processor when the thread is being executed on it. This state typically consists of the program counter, hardware registers, and the execution stack of the thread. As a result, creating a thread also involves the assignment of a valid state to the thread.

Tasks can be created in various ways. However, for the purpose of the channel library we are interested in creating tasks on remote processors. As a result, the remote shell execution program `rsh` can be used (Fig. 2.2). This program takes the name of an executable and a machine as arguments, and spawns a new task with a single thread of execution on the remote host. The `rsh` program comes with the Mach operating system.

2.3 Ports and Messages

Under Mach, tasks (and threads) can communicate with each other by sending messages. This message-passing mechanism is supported through two basic abstractions: *messages* and *ports*.

A message consists of a fixed header—which contains information about the length of the message and its destination—and a list of typed values. Messages are the basic unit of communication. They can be of arbitrary length, and can contain data, pointers, or ports.

In Mach, the destination of a message is specified by a port. Any message sent to a port is stored internally by the Mach kernel. Conceptually, a port is like a mailbox: there are a number of tasks that can insert messages into the port; threads in at most one task can remove messages from the port.

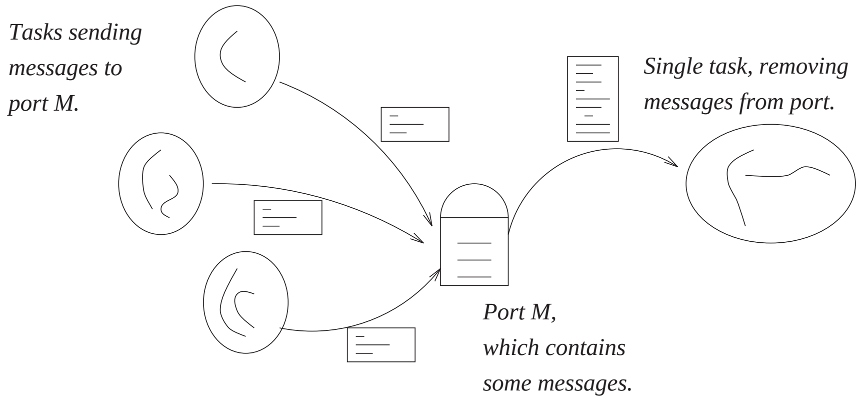
Tasks sending messages to port M.
Single task, removing messages from port.
Port M, which contains some messages.

Figure 2.3: Mach communication ports

2.3.1 Access Rights

Each port has two access rights associated with it: send rights, and receive rights. If a task has send rights for a port, it can send messages to that port. If a task has receive rights for a port, it can receive messages that have been sent to the port. Only one task may have receive rights for a port. Multiple tasks can have send rights for a single port. Note that threads within a task have all the access rights of the task. Therefore, there could be multiple threads which have receive rights for a single port.

Port rights can be granted to a task by sending them to the task in a message. If a message contains a port, then the port rights the task has for that port are sent to the task that receives the message. If send rights are sent in a message, then send rights are kept by the originator of the message as well as passed to the recipient. Since at most one task can have receive rights for a port, if receive rights are sent in a message, the sender no longer has receive rights for the port.

2.3.2 Internals

Internally, a port consists of a finite queue of bounded size which contains the list of messages that have been sent to the port. The size of this queue can be specified for a given port. A task with receive rights can dequeue messages from this internal queue. Sending a message to a port appends the message to the queue associated with the port. Sending a message to a port whose queue is full will result in the suspension of the sender.

Ports can refer to tasks on the local machine or tasks across the network. The interface to the ports is exactly the same. The only difference is that ports over the network are handled (transparently) by the Mach network server. The network server implements *network ports*, which represent ports on remote machines. These ports are used by tasks to send messages to remote machines.

Mach provides a port server known as the *network name server*. This server enables a task to associate a port with a name. Other tasks can then look up the port of interest using the name associated with the port. This name server is used to

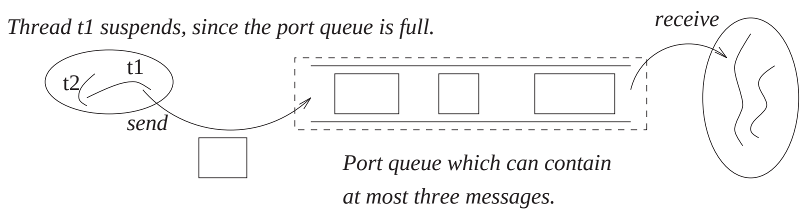
Thread t1 suspends, since the port queue is full.
receive
t1
t2
send
Port queue which can contain
at most three messages.

Figure 2.4: Mach ports—Internal queue

register a service, which then allows other tasks to use the service by looking up the service port from the name server. For instance, if an application programmer decided to write a server that provided access to a remote file system, then the programmer could register this new server with the network name server by giving it a name. Now any application that wants to use this new service can look up the port for the file system server and then send requests for remote files to the server using the port it receives from the name server.

2.4 C-Thread Library

The C-thread library consists of functions that are used to manipulate and synchronize threads [3]. This library provides an interface to the Mach kernel services for thread creation and synchronization. The following three main classes of functions are provided by the library:

1. Functions that are used to manipulate threads.
2. Mutex variables, and functions to manipulate them.
3. Condition variables, and functions to manipulate them.

Synchronization in the C-thread library is done using *condition* and *mutex* variables.

2.4.1 Thread Functions

The thread functions can be used to create, terminate, and schedule threads. These functions call existing Mach functions with the appropriate parameters.

The main advantage of using the thread functions from the C-thread library rather than the ones provided by Mach is that the C-thread library is portable across architectures. As mentioned earlier, creating a thread in Mach involves setting up the state for the thread. This involves assigning values to hardware registers, which differ from machine to machine. The C-thread library can be used to spawn a thread that starts execution from a function that takes a single argument. All the information required to set up the state can be determined by knowing the C compiler on the system, and the address of the function being spawned.

2.4.2 Mutex Variables

Mutex (mutual exclusion) variables are used to protect access to shared data structures. A mutex variable can be in one of two states: locked or unlocked. A thread attempting to lock a mutex variable will suspend if the mutex is in the locked state. If the mutex variable was unlocked, the mutex will be locked, and execution of the thread will continue. Unlocking a mutex will resume execution of one thread that was suspended due to a lock operation on the mutex.

The following functions operate on mutex variables:

`mutex_lock(m)` is used to lock the mutex variable `m`.

`mutex_unlock(m)` is used to unlock the mutex variable `m`.

Initially, the mutex variable is unlocked.

Consider the following example:

```
...
mutex_lock(m);
modify data structure;
mutex_unlock(m);
...
```

If all access to the data structure are protected with the mutex `m` as above, then we know that at most one thread can modify the data structure at a time.

2.4.3 Condition Variables

Condition variables are used when a thread wants to wait for another thread to complete some action. Every condition variable must be associated with a mutex variable. Two basic operations on condition variables are allowed: *signal* and *wait*.

A call to function `condition_wait(c,m)` does the following actions:

1. The mutex `m` is unlocked.
2. The thread is suspended waiting for a signal on the condition variable `c`.
3. On receiving a signal, the mutex `m` is locked.

`condition_signal(c)` wakes up a single thread (if any) waiting on a condition variable. Condition variables can be used to selectively wake up threads that might be suspended waiting for some boolean condition to hold.

Chapter 3

Specification

3.1 Preliminaries

The channel library provides the following three basic abstractions:

- Processes.
- Threads.
- Channels.

A process can be thought of as a virtual address space. A process is similar to a Mach task. The major difference between Mach tasks and processes is that processes do not share memory. Therefore, two processes cannot interact through shared data structures. The concept of a thread is the same as the one in Mach. Threads run within a process, and they share data with other threads in the same process.

The major departure from the Mach model is in the abstraction provided for communication. The channel library provides abstract communication streams known as *channels* (Fig. 3.1) which are used for communication and synchronization. These channels can be thought of as first-in-first-out (FIFO) message buffers.

Three operations can be performed on a channel:

1. Send message along channel.

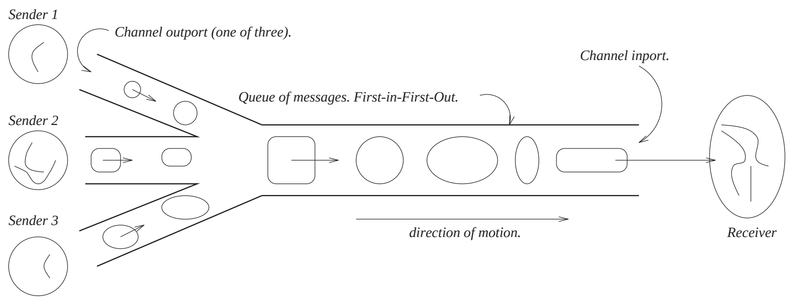
Sender 1
Channel outport (one of three).
Channel inport.
Queue of messages. First-in-First-Out.
Sender 2
Sender 3
direction of motion.
Receiver

Figure 3.1: Channel

2. Receive message from channel.
3. Close channel.

Sending a message along a channel is equivalent to inserting the message into the FIFO. Receiving a message from a channel is equivalent to removing the next message from the FIFO. The close channel operation is somewhat different in that it inserts a special message into the buffer. This message indicates that no more messages will be received along the channel. Furthermore, the sender cannot send a message along the channel after closing the channel. Once a receiver removes the close channel message from a channel, the resources used by the channel can be released.

A channel can have any number of senders, but at most one receiver. If a channel has more than one sender, then the channel is closed only when all the senders close the channel.

To be able to send (receive) data along a particular channel, a process must have *send* (*receive*) *rights* for the channel. These rights are similar to the send and receive rights for Mach ports. However, these rights cannot be passed from one process to another. Once the receiver for a channel is specified, the receiver cannot be changed. However, since a channel could have many senders, different processes could be given send rights for the same channel.

A send operation blocks only when no process has receive rights for the channel. A receive operation blocks only when there is no message in the channel. Once some process has receive rights for a channel, a send operation on that channel never blocks.

The *inport* of a channel is the receiving end of the channel, and the *outport* is the sending end.

The channel library comes with a program known as the network channel server. This program must be running on all the machines in the computation. Chapter 4 describes the purpose of this server in detail.

3.1.1 Conventions

The *main process* in a concurrent computation refers to the first process that is created in the computation. This process is responsible for creating any additional processes in the computation.

The channel library can be specified by means of the functions it provides to the user. Note that the specification is given only in terms of predicates on state, as is usually done for a sequential program. However, the proof of the specification will involve the fact that there could be concurrent access to each function in the library. Each function in the channel library is specified by the following:

- Function is the name of the function, along with its arguments and their types. C syntax is used for the argument list.
- Precondition, Postcondition. If the function is executed in a state in which the precondition holds, then the function will terminate, and on termination, the postcondition will hold.

- Wait Condition. This gives the condition under which execution of the function will suspend. If the wait condition is omitted, the function will never suspend.
- Error Condition is the condition which will cause the function to terminate with an error status. If the error condition is omitted, the function will never terminate with an error status.

The channel library functions do not return any values.

3.2 Initialization

The initialization functions are used to set up the internal data structures that are used by the channel library. One of these functions should be called once per process before calling the channel library functions.

Function: `Initialize_Main ()`
Precondition: The current process is the main process. No other function from the channel library has been called.
Postcondition: The channel library has been initialized.
Error Condition: Duplicate call to `Initialize_Main()`.

Function: `Initialize_Process ()`
Precondition: The current process is not the main process. No other function from the channel library has been called from the current process.
Postcondition: The channel library has been initialized in this process.
Error Condition: Duplicate call to `Initialize_Process()` by the same process.

3.3 Channels

The channel functions are used to create channels, and to manipulate send and receive rights for a channel.

Function: `New_Channel (int *`*chan*`)`
Precondition: The channel library has been initialized in the calling process. *chan* is a non-`NULL` valid pointer to an integer.
Postcondition: $*chan$ contains the logical name of new channel which has no sender and no receiver.
Error Condition: The channel library has not been initialized.

The following two functions are used by a process to obtain access rights for a particular channel. These functions must be called before actually sending or receiving data along a channel.

Function: `Sending_On (int` *chan*`)`

Precondition: *chan* is the logical name of a channel. The calling process does not hold send rights for the channel *chan*.

Postcondition: Send rights for the channel *chan* have been requested by the calling process.

Error Conditions: *chan* is not a valid channel. Initialization has not been done. The calling process already has send rights for channel *chan*. *chan* has been closed by some process.

Function: `Receiving_On (int` *chan*`)`

Precondition: *chan* is the logical name of a channel. The calling process does not hold receive rights for the channel *chan*.

Postcondition: Receive rights for the channel *chan* have been requested by the calling process.

Error Conditions: *chan* is not a valid channel. Initialization has not been done. The calling process already has receive rights for channel *chan*. Another process in the computation has receive rights for channel *chan*.

For instance, the following code in a process would create a new channel, and then declare that the calling process would like to send a message on the newly created channel.

```
int chan;
...
New_Channel (&chan);
Sending_On (chan);
...
```

3.4 Communication

The following functions are used to perform the communication actions discussed earlier. They use a format specification—similar to the one used by the C `printf` function—to specify the type of the message being sent. The format specifications that have been implemented are: `%d` for integers and `%f` for floating-point numbers. Spaces are not permitted in the format specification.

Function: `Send (int` *chan*`, char *`*fmt*`, ...)`

Precondition: *chan* is the logical name of a channel. The calling process has send rights for the channel *chan*. *fmt* is a valid format specification. The argument list following *fmt* matches the format specification.

Wait Condition: There is no receiver for the channel *chan*.

Postcondition: The specified message has been sent to the receiver for channel *chan*.

Error Conditions: The calling process does not have send rights for channel *chan*. The channel *chan* has been closed. *fmt* is not a valid format specification.

Function: `Receive (int` *chan*`, int *`*closed*`, char *`*fmt*`, ...)`

Precondition: *chan* is the logical name of a channel. The calling process has receive rights for the channel *chan*. *fmt* is a valid format specification. *closed* is a valid non-`NULL` pointer. The argument list following *fmt* are pointers to data structures which match the types specified in the format specification.

Wait Condition: There is no message in channel *chan*.

Postcondition: The next message from the channel (if any) has been received, and copied into the pointers in the list. If the channel was closed, then **closed* is one; otherwise **closed* is zero.

Error Conditions: The calling process does not have receive rights for channel *chan*. The channel *chan* has been closed, and an earlier receive operation on the same channel had **closed* set to one. *fmt* is not a valid format specification. The received message length is not the same as the length specified by the format specification.

Notes: Notice that a receive will allow typecasting of equal length messages. However, this is not recommended since the results are machine-dependent.

Function: `Close_Channel (int` *chan*`)`

Precondition: *chan* is the logical name of a channel. The calling process has send rights for channel *chan*.

Wait Condition: There is no receiver for channel *chan*.

Postcondition: This outport for the channel has been closed.

Error Conditions: *chan* is not a valid channel identifier. The channel *chan* was closed. The calling process does not have send rights for channel *chan*.

For instance, a process might send ten messages along a channel, and then close it as follows:

```
int i, chan;
...
Sending_On (chan);
...
for (i=0; i < 10; i++)
    Send (chan, "%d", i*i);
Close_Channel (chan);
...
```

3.5 Processes and Threads

The following functions are used to create and terminate processes and threads.

Function: `Spawn_Process (char *`*proc*`, char *`*host*`, int` *n*`, ...)`

Precondition: The channel library has been initialized. *proc* is a valid non-`NULL` pointer. *host* is a valid non-`NULL` pointer. *proc* specifies the pathname of an executable on *host*. *n* is the number of arguments following *n*, which are channel identifiers.

Postcondition: The process *proc* has been spawned on host *host*, with arguments specified by the channel identifiers following *n* in the argument list of `Spawn_Process()`.

Error Conditions: The remote shell program was not found. The *host* does not allow remote shell access for the user without a password. There are too many processes on the current host.

Notes: To allow remote shell access without a password, you must have a `.rhosts` file in your home directory. See the man page for `rsh` on your system on how to set up this file.

Function: `Spawn_Thread (void (*`*f*`)(), int` *n*`, ...)`

Precondition: *f* is a valid non-`NULL` pointer. *n* is the number of arguments following *n*, which are channel identifiers. *n* is less than `MAXARGS` which is defined to be 10.

Postcondition: A new thread has been spawned within the current task, starting in the function specified by *f*. The function was called with arguments specified by the list of arguments after *n*.

Error Condition: The number of arguments exceeded the maximum limit.

Function: `Finished()`

Precondition: true.

Postcondition: The thread has terminated.

Notes: This function terminates the main thread of control. This function is called once per task.

For instance, the following call would create a new process `nproc` on machine `pete`, with two channel arguments `ch1` and `ch2`.

```
...
Spawn_Process ("proc", "pete", 2, ch1, ch2);
...
```

Chapter 4

Implementation

The implementation has been split into two parts:

1. The network channel server.
2. The user interface.

We begin by discussing the issues involved in implementing the channel library. The network channel server is discussed in detail, followed by the user interface. A code outline of the implementation is given for both the channel server and the user interface.

4.1 Issues

4.1.1 Channels

Mach provides ports for communication. One possible implementation is to use a single port for each channel. However, ports are system-wide resources that are maintained by the Mach kernel. To conserve ports, the implementation uses a single port for each process which is used to receive messages for all channels connected to that process. We now need a mechanism to distinguish between messages being sent to a process. This is done by tagging each message with an identifier which specifies which channel the message was sent on.

For this strategy to succeed, we have to ensure that these message tags are unique across the entire computation. The network channel server is used to provide an identifier that is unique to a machine. As a result, the pair (hostname, identifier) is unique across the computation. This pair is used to identify a channel.

Since a single Mach port is used for a number of channels, each process must queue messages for a single channel internally. Each process has an internal message handling thread that is used for this purpose.

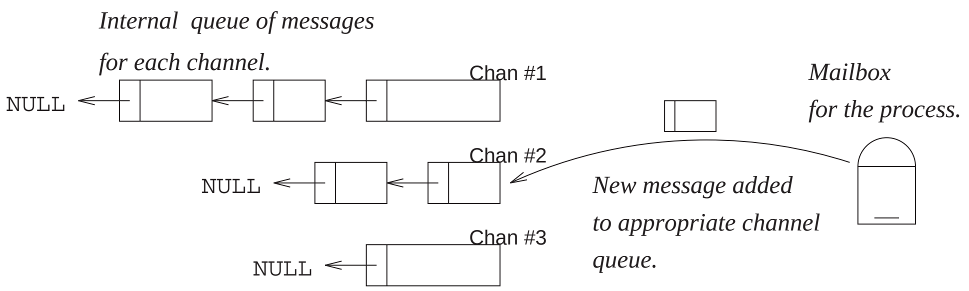

Figure 4.1: Implementing Channels

4.1.2 Threads and Processes

Threads are implemented on top of the C-thread package. This package allows us to spawn a function with a single argument. Since `Spawn_Thread()` takes a variable number of arguments, we have to implement an argument passing mechanism for newly created threads.

The `rsh` program is used to create processes. However, since this program is UNIX based, it does not give us any mechanism to determine the Mach port of the newly created process. Therefore, argument passing for processes must be handled without using Mach messages.

4.1.3 Termination and Channel Close

The C-thread package ensures that when all the threads in the computation terminate, the task itself terminates. However, the implementation uses an extra message handling thread that is always suspended waiting for a message. To be able to terminate gracefully, the channel library keeps track of the number of threads created by the system. Every time a new thread is created, a counter is incremented. When a thread terminates, the counter is decremented. When the number of threads in the system created by the user is zero, the process has terminated. The message handling thread is aborted by means of a function provided by the C-thread library, and the process terminates since all the C-threads have terminated.

For a single sender channel, closing a channel is equivalent to sending a special message on the channel. However, the channel close operation becomes complicated once the number of senders is more than one. The receiver of a channel does not know the number of senders that it has. Therefore, some scheme must be used to detect the number of senders, and to close the channel only when all the senders terminate. The network channel server is the only process that knows the number of senders on a channel. This process is used to send a special close-channel message once the channel has been closed.

4.2 Network Channel Server

The network channel server has three basic purposes:

1. It provides unique channel identifiers to all processes on a single machine.
2. It grants send and receive rights for channels.
3. It is used to close channels once all the senders have closed the channel.

A copy of this server must be executing on every machine in the computation.

Each channel is created by a specific network channel server. The network channel server maintains a list of channels it has created and their current status. The following user functions interact with the channel server:

- `New_Channel()`
- `Sending_On()`
- `Receiving_On()`
- `Close_Channel()`

`New_Channel()` sends a message to the channel server requesting a new channel identifier. The channel server returns an identifier it has not used before. `Sending_On()` requests send rights for a particular channel from the server that created it. `Receiving_On()` requests receive rights for a particular channel from the server that created it. `Close_Channel()` sends a message to the channel server indicating that the sender has closed the channel. Note that channel requests are sent to the server that originally created the channel.

4.2.1 Channel Table

The network channel server maintains a table that contains the status of all the channels it created. Each channel entry is of type `channel_t`, which is a structure defined as:

```
typedef struct {
   int channel_id;
   unsigned int oneclosed:1;
   unsigned int closed:1;
   unsigned int nsent;
   unsigned int nout;
   unsigned int in_registered:1;
   unsigned int in_done:1;
   unsigned int out_registered:1;
   unsigned int out_done:1;
   port_t inport;
   port_t outport;
   mutex_t m;
   condition_t c;
} channel_t;
```

`channel_id` is an integer which is unique to this channel server process. It is used to distinguish between channels.

The flag `oneclosed` is true if any sender for the channel has performed a close channel operation. Once any sender closes the channel, the server no longer grants send rights for the channel to any other process.

`closed` is true just when the channel has been closed. To detect the end-of-channel, two fields are used: `nout` is the number of senders for the channel who have not performed a channel close operation; `nsent` is the number of messages that were sent by the senders that have already closed the channel.

`in_registered` is true when the receiver for the channel is known. The Mach port of the receiver process is stored in `inport`.

`out_registered` is true when some sender for the channel is known. The Mach port of the first sender process is stored in `outport`.

`in_done` is true when receive rights for the channel have been granted to some process.

`out_done` is true when send rights for the channel have been granted to a process.

The mutex `m` and condition variable `c` are used to control access to the channel fields.

4.2.2 Log File

Each channel server creates and writes information out to a log file. This file records the result of operations for every channel handled by the channel server, and any errors that might be returned by the Mach operating system. The log file can be used to aid debugging, since the status of every channel in the computation is recorded in this file. The following messages are logged by the server:

```
-CTRL-START-
-CTRL-CLEAR-
-CTRL-END-
## Terminated ##
```

These three messages are used to report the status of the server itself. The first three messages denote the type of the message that was received by the server. The last message is used to denote that the server was terminated normally. These messages can be used to determine the type of requests received by the server.

The following messages denote various channel operations that were successful:

`[chan=#] registered.` This message is denotes that fact that the specified channel number was returned by the server to some user process. The log file contains such a message whenever a new channel is created by the user.

`closed channel [#].` This message indicates that a close channel request was received for the channel. Note that it does not mean that the channel was actually closed, since the channel could have more than one sender.

`INport [chan=#] recvd.` This message indicates that some process requested receive rights for the specified channel. Note that the receiver could be a process on the local machine or a remote machine.

`OUTport [chan=#] recvd.` This message indicates that a process requested send rights for the specified channel.

`>>warning:: [chan=#] multi-send.` This message indicates that there is more than one sender for the specified channel.

The messages mentioned above can be used to determine the status of the channels created by the user. Note that the channel identifier in the log file is not the same as the channel identifier used in each process. However, it is the same as the identifier printed out by any error message you may receive.

An error message of the form

operation `[chan=#] -` *error message*

indicates that some server operation failed. The possible error messages that you can get are:

```
duplicate registration of inport
multiple requests for an inport
multiple requests for an outport
specified channel was already closed
duplicate registration of channel
channel not found
```

4.2.3 Messages

The network channel server interacts with the user interface functions via messages. There are two types of messages exchanged between the user interface and the server:

- request messages.
- reply messages.

The request messages are sent by the user interface to the channel server. These messages have type `request_t`, defined below:

```
typedef struct {
    msg_header_t h;
    msg_type_t ct;
    int type;
    msg_type_t t0;
    int channel_id;
    int chan_type;
    msg_type_t t1;
    port_t port;
} request_t;
```

The `msg_header_t` and `msg_type_t` are fields required by the Mach messaging system. `msg_header_t` specifies the length of the message, the destination of the message, and the sender of the message. The `msg_type_t` fields are used to specify the types of the fields following them in the structure.

`type` The type field is used to indicate what kind of request is being made. The following message types are recognized by the server:

- `CTRL_START` is used to restart the channel server.
- `CTRL_END` is used to terminate the channel server.
- `CTRL_CLEAR` is not used by the user interface. It is used to kill suspended threads.
- `CTRL_REGISTER` is used to create a new channel.
- `CTRL_NORM` is used to request send or receive rights for a channel. It is also used to close a channel.

`channel_id` is used to specify the particular channel being referred to by the message. Note that since the channel names created by a single channel server are unique, the hostname is not required to identify the channel.

`chan_type` determines whether the message refers to the inport or to the outport for the specified channel.

`port` is used to grant send rights for the receiver port when sending a receive rights request. These send rights can then be granted to any sender for the specified channel.

Some of the messages expect a response. These responses by the server are of type `reply_t`, defined below:

```
  typedef struct {
     msg_header_t h;
     msg_type_t t0;
     int status;
     int type;
     msg_type_t t1;
     port_t port;
     msg_type_t t2;
     char hostname[128];
 } reply_t;
```

`status` is used to indicate whether the request that triggered this reply message succeeded.

`type` is used to indicate whether the reply refers to an inport or an outport.

`port` is used to grant send rights for the Mach port of a receiver to a process that requested send rights for the channel.

`hostname` is used to identify the channel uniquely. Notice that this field is required for the reply message since there might be more than one channel server sending messages to a user process.

4.2.4 Code Outline

Note: Underlined functions are potential points at which the executing thread might suspend.

Function F4.0: `server_start()`

```
initialize data structures;
do
   receive request message.
   if  request.type = CTRL_START → kill all threads;
                                   clear all tables;
   [] request.type = CTRL_END   → kill all threads;
                                   terminate;
   [] request.type = CTRL_REGISTER → spawn handle_register_request;
   [] request.type = CTRL_NORM → spawn handle_regular_request;
   [] request.type = CTRL_CLEAR → kill all threads;
   [] otherwise → log error;
   fi
od
```

Function F4.1: `handle_register_request()`

```
err = register_channel;
if  channel id = -1 → send reply message with new id
[]  otherwise → skip
fi
cthread_exit;
```

Function F4.2: `handle_regular_request()`

```
if  request = close  → err = close_channel;
[]  request = inport  → err = register_inport;
                         err = get_outport;
                         send reply message;
[]  request = outport → err = register_outport;
                         err = get_inport;
                         send reply message;
[]  otherwise → log error
fi
cthread_exit;
```

Function F4.3: `register_channel()`

```
mutex_lock(chan_lock);
channel id := next_valid_chan++;
create new table entry;
mutex_unlock(chan_lock);
```

Function F4.4: `find_channel()`

```
mutex_lock(chan_lock);
if  channel in table → mutex_lock(channel.m);
                       mutex_unlock(chan_lock);
```

```
                  return channel;
    ▯  otherwise  → mutex_unlock(chan_lock);
                  return NULL;
    fi
```

Function F4.5: `close_channel()`

```
    chan := find_channel;
    if  chan = NULL  → error ▯ otherwise  → skip fi
    if  chan.closed  → error ▯ otherwise  → skip fi
    chan.nsent += msgs;
    chan.oneclosed := 1;
    chan.nout–;
    if  chan.nout = 0  → chan.closed := 1;
                  send close channel message;
                  condition_broadcast(chan.c);
    ▯  otherwise  → skip
    fi
    mutex_unlock(chan.m);
```

Function F4.6: `register_inport()`

```
    chan := find_channel;
    if  chan = NULL  → error ▯ otherwise  → skip fi
    if  chan.in_registered  → error ▯ otherwise  → skip fi
    chan.in_registered := 1;
    condition_signal(chan.c);
    mutex_unlock(chan.m);
```

Function F4.7: `register_outport()`

```
    chan := find_channel;
    if  chan = NULL  → error ▯ otherwise  → skip fi
    chan.nout++;
    if  chan.out_registered  → warning ▯ otherwise  → skip fi
    chan.out_registered := 1;
    condition_signal(chan.c);
    mutex_unlock(chan.m);
```

Function F4.8: `get_inport()`

```
    chan := find_channel;
    if  chan = NULL  → error ▯ otherwise  → skip fi
    if  chan.oneclosed  → error ▯ otherwise  → skip fi
    do ¬chan.in_registered  →
      condition_wait(chan.c,chan.m);
      if  chan.oneclosed  → condition_signal(chan.c);
                 mutex_unlock(chan.m);
                 error;
      ▯  otherwise  → skip
      fi
    od
    condition_signal (chan.c);
    **if** *chan.oneclosed* $\rightarrow$ *error* [] *otherwise* $\rightarrow$ *skip* **fi**
    **if** *chan.out_done* $\rightarrow$ *error* [] *otherwise* $\rightarrow$ *skip* **fi**
    *chan.in_done := 1;*
    *mutex_unlock(chan.m);*
```

Function F4.9: `get_outport()`

```
    chan := find_channel;
    if chan = NULL → error [] otherwise → skip fi
    if chan.oneclosed → error [] otherwise → skip fi
    do ¬chan.out_registered →
      condition_wait(chan.c,chan.m);
      if chan.oneclosed → condition_signal(chan.c);
                    mutex_unlock(chan.m);
                    error;
      [] otherwise → skip
      fi
    od
    condition_signal (chan.c);
    if chan.oneclosed → error [] otherwise → skip fi
    if chan.out_done → warning [] otherwise → skip fi
    chan.out_done := 1;
    mutex_unlock(chan.m);
```

4.3 User Interface

The user interface consists of the functions that were specified in chapter 3. These functions use the network channel server to create and close channels, and to request send or receive rights for channels.

4.3.1 Channel Table

Each process maintains a table of known channels. This table consists of elements of the following type:

```
typedef struct {
   int channel_id;
   char *host;
   port_t inport;
   port_t outport;
   unsigned int is_inport:1;
   unsigned int in_done:1;
   unsigned int is_outport:1;
   unsigned int out_done:1;
   unsigned int closed:1;
   unsigned int rclosed:1;
   unsigned int nsent;
   unsigned int nrecv;
```

```
      list_t *list_of_msgs;
      int nmsgs;
      struct mutex m;
      struct condition c;
   } chan_tab_t;
```

`channel_id` and `host` together uniquely identify the channel name.

`inport` and `outport` are the Mach ports of the sender and receiver.

`is_inport` indicates whether or not the process has requested receive rights for the channel. `in_done` is used to indicate whether the process has received receive rights for the channel. `is_outport` and `out_done` are similar but used to determine the status of an outport.

`closed` is used to determine if the channel is closed, i.e., if all senders have closed the channel. `rclosed` is used to determine if at least one sender has closed the channel. `nsent` and `nrecv` are used to detect if the channel was closed.

`list_of_msgs` points to a queue of messages in the channel. `nmsgs` is the number of messages in the queue.

`m` and `c` are variables used for synchronization.

4.3.2 Message Handling

Each message that is sent along a channel contains the message itself, and the unique channel name. The message structure used is:

```
   typedef struct {
      msg_header_t h;
      msg_type_t t;
      char *msg;
      msg_type_t ht;
      char host[128];
   } generic_msg_t;
```

`msg` points to the data contained in the message. `host` is used to identify the channel.

The `msg` field contains all the data that is sent in the message. Each data item specified by the format specification in a `Send` is copied into a single buffer. Note that the virtual memory functions of Mach are used to allocate this buffer. As a result, when such a message is sent to a process on the local machine, Mach will not make a second copy of the buffer, since Mach uses a copy-on-write scheme for managing virtual memory.

The `list_of_msgs` field in the channel table is used to store the messages that have been received for a particular channel. Each process has a single thread in it that is responsible for receiving messages for all channels for which the process has receive rights. Each channel has a count `nmsgs` associated with it, that refers to the number of messages that were received for the channel. To queue a message for a channel, the following operations are done:

1. The channel mutex `m` is locked.
2. The received message is appended to the `list_of_msgs`.
3. A signal action is performed on the condition variable `c`.
4. `nmsgs` is incremented.
5. The channel mutex `m` is unlocked.

To remove a message from the channel, the following actions are done:

1. The channel mutex `m` is locked.
2. A wait is done on the condition variable `c` until `nmsgs` is non-zero.
3. The next message is removed from the queue.
4. `nmsgs` is decremented.
5. The channel mutex `m` is unlocked.

4.3.3 Argument Passing

When a `Spawn_Thread()` or a `Spawn_Process()` call is made, the list of arguments to the thread or process must be converted into a form that can actually be passed to the thread or process, and then reconverted back into the original arguments. Since process and thread creation are very different internally, two separate strategies are used to tackle the problem for each of them.

Threads

The C-thread package allows a thread to be spawned with a single argument. Therefore, we have to convert the list of arguments to `Spawn_Thread()` into one single structure that can then be passed to some function that converts the structure back into the original argument list. The structure used for this purpose is shown below:

```
struct single_argument {
   void (*f)();
   int n;
   int *chan_list;
};
```

`n` is the number of integers in the array `chan_list`. Together, these specify all the arguments that were passed to the thread. `f` is a pointer to the function that is to be spawned.

`Spawn_Thread()` does not spawn the function specified in its argument list. Instead, a function known as `__generic_unpack` is spawned. This function takes a single argument of type `struct single_argument*`. The purpose of this function is to unpack the arguments and call the function specified by `f` with the appropriate argument list.

Processes

Argument passing across processes is slightly more complex. The major problem is that the channel identifiers that are passed as arguments do not refer to the channel name, but to the location of the channel in the channel table. Therefore, the channels must be entered into the channel table in the remote process, and then converted into a valid channel identifier in the remote process. The channel names are passed using the command-line mechanism. This list must then be unpacked by the remote process, and entered into the channel table.

The channel library provides a simple tool known as `makestub` which is used to create the main process for remote processes that are spawned using `Spawn_Process()`. The main program takes the command line arguments, registers them internally, and then calls a specified function (an argument to `makestub`) with the actual channel identifiers.

4.3.4 Code Outline

Note: Underlined functions are potential points at which the executing thread might suspend.

Function F4.10: `internal_msg_handler()`

```
do
  receive message
  if user interrupt → cthread_exit() ▯ otherwise → skip fi
  chan := find_channel()
  if chan = NULL → mutex_lock(table_mutex);
        insert new channel into table;
        chan := new channel;
        mutex_unlock(table_mutex);
  ▯  chan ≠ NULL → skip
  fi
  mutex_lock(chan.m);
  insert message in message queue;
  condition_signal(chan.c);
  mutex_unlock(chan.m);
od
```

Function F4.11: `Sending_On()`

```
if chan.is_outport → error ▯ otherwise → skip fi
chan.is_outport := 1;
chan.out_done := 0;
```

```
    send outport request to channel server
```

Function F4.12: `Receiving_On()`

```
    if chan.is_inport → error ▯ otherwise → skip fi
    chan.is_inport := 1;
    chan.in_done := 0;
    send inport request to channel server
```

Function F4.13: `Initialize_Main()`

```
    look up channel server;
    initialize all tables;
```

Function F4.14: `Initialize_Process()`

```
    initialize all tables;
```

Function F4.15: `New_Channel()`

```
    request new channel id from server;
    if channel exists → error ▯ otherwise → skip fi
    $\underline{mutex\_lock(table\_mutex);}$
    create new table entry;
    mutex_unlock(table_mutex);
```

Function F4.16: `Send()`

```
    if $\neg$chan.is_outport → error ▯ otherwise → skip fi
    if $\neg$chan.out_done → complete_orpc() ▯ otherwise → skip fi
    $\underline{mutex\_lock(table\_mutex);}$
    if chan.closed → error ▯ otherwise → skip fi
    mutex_unlock(table_mutex);
    package message into buffer;
    send message
```

Function F4.17: `Receive()`

```
    if $\neg$chan.is_inport → error ▯ otherwise → skip fi
    if $\neg$chan.in_done → complete_irpc() ▯ otherwise → skip fi
    $\underline{mutex\_lock(chan.m);}$
    if channel closed → error ▯ otherwise → skip fi
    do chan.msgs = 0 → $\underline{condition\_wait(chan.c,chan.m)}$ od
    take next message out of queue;
    unpack message;
    mutex_unlock(chan.m);
```

Function F4.18: `Spawn_Thread()`

```
    pack arguments into one structure;
    $\underline{mutex\_lock(thread\_mutex);}$
    thread_count++;
    mutex_unlock(thread_mutex);
    spawn new thread
```

Function F4.19: `Spawn_Process()`

```
    create command line arguments
    use rsh to spawn new process
```

Function F4.20: `Finished()`

```
    mutex_lock(thread_mutex);
    thread_count–;
    if thread_count = 0 → abort internal_msg_handler() ▯ otherwise → skip fi
    mutex_unlock(thread_mutex);
    thread exit;
```

Function F4.21: `Close_Channel()`

```
    if chan.closed → error ▯ otherwise → skip fi
    chan.closed := 1;
    send close channel message to server;
```

Function F4.22: `complete_irpc()`

```
    mutex_lock(rpc_mutex);
    do ¬chan.in_done →
        wait for reply message from server;
        enter message into table;
        set done flag for appropriate channel;
        mutex_unlock(rpc_mutex);
        mutex_lock(rpc_mutex);
    od
    mutex_unlock(rpc_mutex);
```

Function F4.23: `complete_orpc()`

```
    mutex_lock(rpc_mutex);
    do ¬chan.out_done →
        wait for reply message from server;
        enter message into table;
        set done flag for appropriate channel;
        mutex_unlock(rpc_mutex);
        mutex_lock(rpc_mutex);
    od
    mutex_unlock(rpc_mutex);
```

4.4 Possible Improvements

The following modifications could be made to the Network Channel Server. These modifications, in some cases, incur a cost which a user may or may not be willing to pay.

- Each server could have a `CTRL_SESSION` message, which returns a new session id. The channel would then be uniquely identified by the triple (hostname, session id, channel id). This would allow multiple parallel applications to use the channel server.

- The network channel server could be simply removed and incorporated into the internal message handling thread in the user interface. However, this presents the additional cost of registering all user processes with the network nameserver.

Chapter 5

Proof Outline

This chapter contains an *informal* proof of the channel library. Since there are two basic parts to the channel library, we split the proof into two parts:

1. Proof of the network channel server.
2. Proof of the user interface functions.

We shall first prove some basic properties of the network channel server. These properties, along with the properties of the user interface functions will demonstrate that the user interface functions satisfy their specification.

5.1 Network Channel Server

We begin by defining the ghost variable *ns*. *ns* is the number of times the channel table is cleared. We define the interval between two successive values of *ns* as a session.

5.1.1 Properties

Property P5.1: *For every channel c, the following pairs are monotonic under lexicographic ordering:* $(ns, c.oneclosed)$, $(ns, c.closed)$, $(ns, c.in\_registered)$, $(ns, c.in\_done)$, $(ns, c.out\_registered)$, $(ns, c.out\_done)$.

Proof: There are two possible values these channel fields could have: 0 or 1. They are assigned a value of zero just when the channel table is cleared, which increases *ns*. All other modifications to the channel fields given above only assign them the value 1.

Property P5.2: *All mutex variables used are created exactly once.*

Proof: The only statements that can modify a mutex variable are the initialization and termination routines. The initialization routine is called exactly once.

Property P5.3: *There is exactly one thread in the channel server until the first receive is done in (F4.0).*

Proof: (F4.0) begins the channel server. This function is the only one that spawns new threads. These spawn operations follow a receive operation.

Property P5.4: *Only threads from the channel server can modify the data structures in the server.*

Proof: The channel server is created as an independent process. Furthermore, there are no system calls in the server that grant memory access to any other task in the system.

Property P5.5: *At most one thread can modify or read the channel table at a time.*

Proof: Initially, there are no threads in the system. From (P5.2) we know that the mutex variables themselves do not change in value, but only in state. The only places where the channel table is modified or read are in functions (F4.4), (F4.3), and (F4.0). From (P5.3), and the fact that the channel table is modified in (F4.0) before the first receive operation, the invariant is maintained. The two functions (F4.4) and (F4.3) lock the mutex variable *chan_lock* before accessing the channel table. Therefore, the invariant is maintained.

Property P5.6: *The pair* $(ns, next\_valid\_chan)$ *is monotonic under lexicographic ordering.*

Proof: $next\_valid\_chan$ is modified in two places: (i) when the channel table is cleared; (ii) in function (F4.3). Clearing the channel table increases ns. (F4.3) increments the value of $next\_valid\_chan$. This increment is done only when the mutex variable *chan_lock* is locked. Therefore, at most one thread can increment $next\_valid\_chan$ at a time. So, (F4.3) only performs monotonic changes to the pair.

Property P5.7: *The channel identifiers returned by the network channel server in a session are unique.*

Proof: The only place where new channel identifiers are created is in the function (F4.3). $next\_valid\_chan$ is used to create new channel identifiers. From (P5.6), we know that the modifications to $next\_valid\_chan$ are monotonic for a given session. From function (F4.3), we see that $next\_valid\_chan$ is changed every time a new channel is created. From (P5.6), we know that no two values of the variable $next\_valid\_chan$ can be the same.

Property P5.8: *At most one thread can modify an entry in the channel table at a time.*

Proof: Initially, there are no threads in the computation. Apart from function (F4.3), all other functions modify a channel entry only after calling (F4.4). This function locks the mutex m associated with the channel. (F4.3) creates a new channel entry. From (P5.7), we know that these channel entries are identified by unique values. Since (F4.3) returns a new channel and we know (P5.7), no other thread in the computation can access the newly created channel entry.

Property P5.9: *For a channel c, c.in_registered is 1 just when an inport register request has been received for the channel c.*

Proof: *c.in_registered* is assigned a value 1 only in function (F4.6). This function is called only from function (F4.1) when the request was an inport request. (F4.1) is called just when a register request is received. (F4.6) assigns a value 1 to *c.in_registered*.

Property P5.10: *For a channel c, c.out_registered is 1 just when an outport register request has been received for the channel c.*

Proof: Similar to the proof of (P5.9).

Property P5.11: *For a channel c, c.oneclosed is 1 just when a close channel request has been received for the channel c.*

Proof: Similar to the proof of (P5.11).

Property P5.12: *The number of reply messages sent for inport requests without an error status is at most one.*

Proof: A reply message is sent to an inport request only after functions (F4.6) and (F4.9) have been called. (F4.6) returns an error if *in_registered* is 1 for the channel. Since *in_registered* is set to 1 every time (F4.6) is called and we know (P5.9), we can conclude that an error status is returned if (F4.6) is called more than once for the same channel. Since a reply message is sent only after (F4.6) is called, we have established (P5.12).

Property P5.13: *The reply message for an inport request on a channel is sent only after the outport request for the channel has been received.*

Proof: A reply message for an inport request is sent only after the functions (F4.7) and (F4.8) have been called. Function (F4.8) suspends on the condition variable for the channel until *in_registered* has value 1. From (P5.9) we can conclude that *in_registered* is 1 just when the outport request for the channel has been received.

Property P5.14: *The reply message for an outport request on a channel is sent* *only after the inport request for the channel has been received.*

Proof: A reply message for an outport request is sent only after the functions (F4.6) and (F4.9) have been called. Function (F4.8) suspends on the condition variable for the channel until *out_registered* has value 1. From (P5.10) we can conclude that *out_registered* is 1 just when the inport request for the channel has been received.

Property P5.15: *A close channel message is sent to the receiver of a channel only after the number of close channel requests is the same as the number of outport requests.*

Proof: Each outport request increases the value of *nout* for the channel. Each close channel request decreases the value of *nout*. These modifications to *nout* are protected by the channel mutex *m*. A close channel message is sent to the receiver only when *nout* is zero.

5.2 User Interface

We use the following fact about the channel table: when a new channel table entry is created, all the bit-fields for the newly created channel have value zero.

5.2.1 Properties

Property P5.16: *At most one thread can access the channel table at a time.*

Proof: All modification to the channel table are protected by the mutex variable *table_mutex*. As a result, at most one thread will succeed in locking the mutex, from which the property follows.

Property P5.17: *(F4.15) creates new channels with identifiers from the local network channel server.*

Proof: From the code outline of (F4.15), we can see that the function satisfies the property mentioned above. Since we have (P5.16), the channel table update is equivalent to an atomic action.

Property P5.18: *For any channel, an outport request from a process is sent at most once to the network channel server.*

Proof: An outport request is sent only from (F4.11). This function sets the *is_outport* flag to 1. It returns an error status just when *is_outport* has value 1 when the function is called.

Property P5.19: *For any channel, an inport request from a process is sent at most once to the network channel server.*

Proof: An inport request is sent only from (F4.12). This function sets the *is_inport* flag to 1. It returns an error status just when *is_inport* has value 1 when the function is called.

Property P5.20: *Function (F4.11) sends an outport request to the network channel server when it is called for the first time.*

Proof: Follows from the code outline of (F4.11).

Property P5.21: *Function (F4.12) sends an inport request to the network channel server when it is called for the first time.*

Proof: Follows from the code outline of (F4.12).

Property P5.22: *(F4.16) succeeds only if the process has called (F4.11) for the channel.*

Proof: The *is_outport* field of a channel is modified only by (F4.11). Initially, the field has value 0. From the code of (F4.16), it is clear that the function succeeds only when *is_outport* has value 1, i.e. when (F4.11) was called for the channel.

Property P5.23: *(F4.17) succeeds only if the process has called (F4.12) for the channel.*

Proof: Similar to the proof of (P5.22).

Property P5.24: *A process sends a close channel request to the network channel server at most once.*

Proof: A close channel request is sent only by (F4.5). The *closed* field of the channel is set by (F4.5). An error status is returned by (F4.5) if it is called for a channel that has the *closed* field set to 1.

5.3 Outline of Proof

Property P5.25: *Function* `Initialize_Main()` *satisfies its specification.*

Proof: The program (F4.13) for the function is trivial, and the proof is left to the reader.

Property P5.26: *Function* `Initialize_Process()` *satisfies its specification.*

Proof: The program (F4.14) for the function is trivial, and the proof is left to the reader.

Property P5.27: *Function* `New_Channel()` *satisfies its specification.*

Proof: Follows from (P5.17), and (P5.7).

Property P5.28: *Function* `Sending_On()` *satisfies its specification.*

Proof: Follows from (P5.20) and (P5.18).

Property P5.29: *Function* `Receiving_On()` *satisfies its specification.*

Proof: Follows from (P5.21) and (P5.19).

Property P5.30: *Function* `Send()` *satisfies its specification.*

Proof: Follows from (P5.22).

Property P5.31: *Function* `Receive()` *satisfies its specification.*

Proof: Follows from (P5.23).

Property P5.32: *Function* `Close_Channel()` *satisfies its specification.*

Proof: Follows from (P5.24) and (P5.15).

Property P5.33: *Function* `Spawn_Process()` *satisfies its specification.*

Proof: Follows from the code outline (F4.19).

Property P5.34: *Function* `Spawn_Thread()` *satisfies its specification.*

Proof: Follows from the code outline (F4.18), and the fact that modification of $thread\_count$ is protected by the mutex $thread\_mutex$.

Property P5.35: *Function* `Finished()` *satisfies its specification.*

Proof: Modification of variable $thread\_count$ is protected by the mutex variable $thread\_mutex$. This variable is initially 0. (F4.18) increases the value of $thread\_count$ whenever it is called. (F4.20) decreases the value of $thread\_count$ whenever it is called. Therefore, $thread\_count$ being zero is equivalent to the condition that there are no user threads in the system. This condition is used to terminate the process.

Property P5.36: *Channel identifiers in the system are unique.*

Proof: Follows from (P5.7).

We now define send rights and receive rights in terms of the messages exchanged between the channel server.

- Requesting send rights is equivalent to sending an outport request to the channel server.
- Requesting receive rights is equivalent to sending an inport request to the channel server.
- The reply message for an outport request grants send rights to a process.
- The reply message for an inport request grants receive rights to a process.

Now, we have the following properties which needed to be established:

Property P5.37: *There is at most one process in the computation that has receive rights for a channel.*

Proof: Follows from (P5.12).

Property P5.38: *A send succeeds only if the calling process has send rights for the channel.*

Proof: Follows from (P5.22).

Property P5.39: *A receive succeeds only if the calling process has receive rights for the channel.*

Proof: Follows from (P5.23).

Chapter 6

Example programs

In this chapter we discuss some example programs that were implemented using the channel library. The purpose of these examples is to demonstrate how one might use the channel library to write a parallel program. We have chosen the following two examples:

- A distributed sorting program.
- A producer-consumer problem.

6.1 Producer-Consumer

In this example, there are two processes in the computation. One of the processes (known as the producer) sends messages to the other process (known as the consumer). The consumer then prints out the messages it received and terminates.

The function `main()` creates the channel connecting the producer and consumer, spawns the two processes, and then terminates.

```
void main(void)
{
  int ch;
  Initialize_Main ();
```

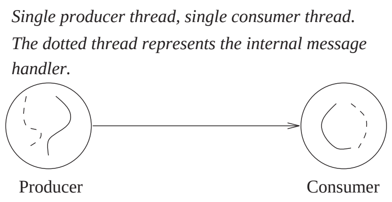
Single producer thread, single consumer thread.
The dotted thread represents the internal message handler.
Producer
Consumer

Figure 6.1: Producer-Consumer Problem

```
  New_Channel (&ch);

  Spawn_Process ("proc/prod", "chainsaw", 1, ch);
  Spawn_Process ("proc/cons", "pistol", 1, ch);
  Finished();
}
```

The producer requests send rights for the channel that was sent to it as an argument, and then sends messages along it. Finally, the channel is closed, and the producer terminates.

```
void producer (int ch)
{
  int i;

  Sending_On (ch);
  for (i=0; i < 10; i++)
        Send (ch, "%d", i*i);
   Close_Channel (ch);
}
```

The consumer requests receive rights for the channel that was sent to it as an argument, and then receives messages from the channel until the close channel message is received.

```
void consumer (int ch)
{
  int i, closed;
  FILE *fp;

  Receiving_On (ch);
  Receive (ch, &closed, "%d", &i);
  fp = fopen ("log", "w");
  while (!closed) {
        fprintf (fp, "%d\n", i);
        Receive (ch, &closed, "%d", &i);
  }
  fclose(fp);
}
```

6.2 Sorting

The program below implements the mergesort algorithm for sorting numbers. This algorithm uses a divide-and-conquer strategy. `main()` is used to create the initial sort process, and to read in the numbers that are to be sorted.

```
void main(void)
```

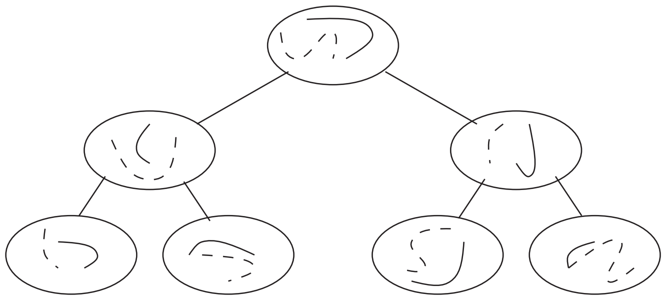

Figure 6.2: Distributed Mergesort

```
{
  int chan1, chan2;
  int i, n, val, closed;

  Initialize_Main();
  New_Channel(&chan1);
  New_Channel(&chan2);
  Sending_On(chan1);
  Receiving_On(chan2);

  Spawn_Process ("mergesort", "pete", 2, chan1, chan2);

  printf ("Enter # of values: ");
  scanf ("%d", &n);
  Send (chan1, "%d", n);
  printf ("Enter values:\n");
  for(i=0; i < n; i++) {
     scanf ("%d", &val);
     Send (chan1, "%d", val);
  }
  Close_Channel(chan1);
  for(i=0; i < n; i++) {
     Receive(chan2, &closed, "%d", &val);
     printf ("%d\n", val);
  }
  Finished();
}
```

Process `mergesort()` checks the problem size. If the base case is reached, then the process sends back the solution and exits. Otherwise, it creates two new processes which are used to solve two sub-problems of the original sorting problem.

```
void mergesort(int in, int out)
{
  int i, n, val;
  int closed;
  int c0, c1, c2, c3;
  int *left, *right;
  Receiving_On (in);
  Sending_On (out);

  Receive(in, &closed, "%d", &n);
  if (n == 1) {
     Receive (in, &closed, "%d", &val);
     Send (out, "%d", val);
  }
  else {
     val = (int*) malloc(sizeof(int)*n);
     New_Channel(&c0); Sending_On (c0);
     New_Channel(&c1); Receiving_On (c1);
     New_Channel(&c2); Sending_On (c2);
     New_Channel(&c3); Receiving_On (c3);
     Send (c0, "%d", (int)n/2);
     for (i=0; i < n/2; i++) {
         Receive (in, &closed, "%d", &val);
         Send (c0, "%d", val);
     }
     Send (c2, "%d", n-(int)n/2);
     for (i=0; i < n-n/2; i++) {
         Receive (in, &closed, "%d", &val);
         Send (c2, "%d", val);
     }
     Close_Channel (c0);
     Close_Channel (c2);
     left = (int*)malloc(sizeof(int)*((int)n/2));
     right = (int*)malloc(sizeof(int)*(n-(int)n/2));
     for (i=0; i < n/2; i++)
         Receive (c1, &closed, "%d", left+i);
     for (i=0; i < n-n/2; i++)
         Receive (c3, &closed, "%d", right+i);
     merge(left, right);
     for (i=0; i < n/2; i++)
         Send (chanout, "%d", left[i]);
     for (i=0; i < n-n/2; i++)
         Send (chanout, "%d", right[i]);
  }
}
```

Chapter 7

Conclusion

We have presented the implementation of a channel library on top of the Mach operating system. From the implementation outline given, it is clear that the channel library implementation was greatly simplified as a result of the support provided by Mach itself. In fact, the entire implementation took 2000 lines of code.

Although implementing the library was a simple task, the proof for the channel library was extremely complicated. Presenting a code outline greatly simplified the proof, by hiding details of the implementation and focusing attention on synchronization issues. In spite of using this simplification, the proof outline was complex.

The key properties in the proof established that certain variables used in the implementation were monotonic. However, all the monotonic variables could be modified in a non-monotonic manner, but only by a single thread of execution. This leads us to believe that reasoning about correctness of the library using shared data structures that are monotonic will greatly simplify the proof.

Reasoning about a concurrent computation in which sharing is restricted to monotonic variables with a single thread of execution that can perform a non-monotonic modification to them can be applied to a large number of existing constructs (for instance, channels, monitors, and locks) used to restrict the interaction between cooperating tasks.

We think that choosing Mach as the base for the channel library was a good decision because of the services provided by the operating system. Proving correctness of the channel library seemed to be a complex task, in spite of the fact that a number of implementation details were hidden.

Bibliography

[1] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., Young, M. *Mach: A New Kernel Foundation for UNIX Development* In *Proceedings of Summer Usenix.* July, 1986.

[2] Bach, M.J. *The Design of the UNIX Operating System.* Prentice-Hall, 1986.

[3] Cooper, E.C., and Draves, R.P. *C Threads* Technical Report CMU-CS-88-154, School of Computer Science, Carnegie Mellon University. February 1988.

[4] Kernighan, B.W., and Ritchie, D.M. *The C Programming Language.* Prentice-Hall, 1988.

[5] NeXT Computer Inc. *NeXTSTEP Operating System Software.* Addison-Wesley, 1992.

[6] Silberschatz, A., Peterson, J., and Galvin, P. *Operating System Concepts.* Addison-Wesley, 1991.