502

The Charlotte Distributed Operating System

Part IV of the First Report on The Crystal Project

Raphael Finkel

Marvin Solomon

David DeWitt

Lawrence Landweber

CONTENTS

1. Introduction to Crystal

The University of Wisconsin Crystal project was funded starting in 1981 by the National Science Foundation Experimental Computer Science Program to construct a multicomputer with a large number of substantial processing nodes. The original proposal called for the nodes to be interconnected using broadband, frequency-agile local network interfaces. Each node was to be a high performance 32 bit computer with a approximately 1 megabyte of memory and floating-point hardware. The total communications bandwidth was expected to be approximately 100 Mbits/second.

During the first year of the project, these specifications have been refined. We have decided to buy approximately 40 node machines, each a VAX-11/750. The interconnection hardware will be the Proteon ProNet. Currently, the ProNet is available in a 10 Mbits/second version. We have contracted with Proteon to increase the effective bandwidth to 80 MBits/second.

1.1. Software Overview

The purpose of this hardware is to promote research in distributed algorithms for a wide variety of applications. In order to provide different applications simultaneous access to the network hardware, we have designed a software package called the *nugget* that resides on each node. In brief, the nugget provides the following facilities:

1. The nugget enforces allocation of the network among different applications by virtualizing communications within partitions of the network. These partitions are established interactively through a host machine.
2. Backing store is shared among the nodes by nugget facilities to virtualize disks.
3. Interaction between the user and individual machines is provided by the nugget facility of virtual terminals.

4. Initial loading, control, and debugging of programs on node machines is controlled by nugget software.

The Charlotte operating system is designed to provide standard interactive operating system support within a Crystal partition. The Charlotte *kernel* provides

1. multiprocessing
2. sophisticated and powerful inter-process communication
3. mechanisms for scheduling, store allocation, and migration.

All policies in Charlotte are concentrated in *utility processes*. They are designed so that each such process controls a policy on its own set of machines. The set may range in size from one machine to the entire partition. The processes that control the same resource on different machine sets communicate with each other to achieve global policy decisions. The utilities that have been designed so far include a switchboard, a program starter (which manages memory), and the file server. In addition, there are non-policy utilities for command interpretation and program connection.

We expect that Crystal will be used for a wide range of applications. Currently research is underway in distributed operating systems, programming languages for distributed systems, tools for debugging distributed systems, multiprocessor database machines, parallel algorithms for math programming, numerical analysis and computer vision, and evaluating alternative protocols for high performance local network communications.

All Crystal software is being written in a local extension to Modula. Our compiler, which runs on a VAX running Berkeley 4.1 Unix, employs syntactic error correction through the FMQ algorithm and is quite fast. The code it generates compares well with that produced by the C compiler.

1.2. Phases of the project

The first phase of the project was dedicated to defining both the hardware and the software. This phase ended in December 1982. Decisions were reached concerning both the node machines and the interconnection devices. The node machine decision was difficult. We had to balance our concerns for reliability, availability, speed, and cost. The machine we chose, the VAX-11/750, although not as fast as others we investigated, had the advantage of being a known architecture for which our Modula compiler already generates code. The Proteon network is currently available. We have been using this network to interconnect our Unix VAX machines and have found it to be extremely reliable.

During the first phase, the nugget was specified and a prototype implementation was completed on a network of eight Digital Equipment PDP-11/23 computers connected by the Megalink CSMA broadband network manufactured by Computrol. Charlotte was also specified and the kernel debugged on this network.

The second phase of the project has just gotten underway. We are finalizing the nugget specifications, which changed in minor ways when we decided that the node machines would be VAXen. The nuggetmaster, which controls the partitions, has also been specified. Charlotte is undergoing debugging of the utility processes. During this phase, which lasts until July 1984, we will transfer the nugget and Charlotte to the node machines and modify them as necessary for the ProNet. Charlotte will be modified to fit with the nugget. (Until now, they have been developed independently.) The utility processes will be supplemented with login and authentication processes, and the file system will be converted to use Crystal disks instead of a file system on the host machine. We plan to have a production, stable operating system by the end of this phase.

The third phase of the project will see large-scale applications actively pursued. Some of this work will start during the second phase. We also expect to re-evaluate the hardware decisions at some point during this phase. There is some reason to expect that frequency-agile modems will be available that will make communication within each partition truly independent of communication within other partitions. Each partition will be able to use its own set of frequencies. Work with optical fiber technology for computer interconnection is also underway at various laboratories around the country. Within five years, impressive bandwidths should be available, reaching into the gigabit/second range. We will continue to monitor progress in this area.

1.3. This report

The purpose of this report is to describe the current state of the design and implementation of the Crystal project. It is intended for readers who have no familiarity with Crystal and wish to see the design decisions that have been made. It is also intended for implementers who need a coherent and reasonably complete specification in order to interface various parts of the project. This dual readership requires us to repeat ideas, first presenting them in an overview fashion, and then diving into tedious details. We urge the reader to skip over those parts of the document that are not at the right level of detail. This report is divided into several documents.

This document describes the Charlotte distributed operating system. It is a direct descendent of the Arachne distributed operating system, which has been discussed elsewhere [1].

---

[1] R. A. Finkel and M. H. Solomon, *The Arachne Kernel, Version 1.2*, University of Wisconsin–Madison Computer Sciences Technical Report 380, April 1980; R. A. Finkel and M. H. Solomon, *The Arachne Distributed Operating System*, University of Wisconsin--Madison Computer Sciences Technical Report 439, July 1981; R. A. Finkel, M. H. Solomon, and R. Tischler, Arachne User Guide, Version 1.2, University of Wisconsin Mathematics Research Center Technical Summary Report 2066, April 1980.

An identical copy of the Charlotte kernel resides on every machine in a partition. The kernel provides a limited number of low-level services. These services can be categorized as inter-process communication, process control, and input-output. A single kernel supports all the processes on its machine. Processes request services by submitting *kernel calls*, which appear much like subroutine calls to a process written in Modula. Some of these processes represent resource allocation policy modules, and should therefore be counted as part of the operating system. We call these modules *utility processes*.

This paper discusses the current design of Charlotte. The kernel for Charlotte has been completely designed. Other parts are still under development. A prototype implementation (based on a closely related, but different design) has been developed and is running on the PDP-11 network, and the current design is currently being debugged on the VAX-11 network, Many utility processes have been defined. Most have been implemented and are being debugged. All Charlotte software is written in Modula, except for startup code written in assembler and some of the interprocess communication code written in C.

2. Inter-process communication

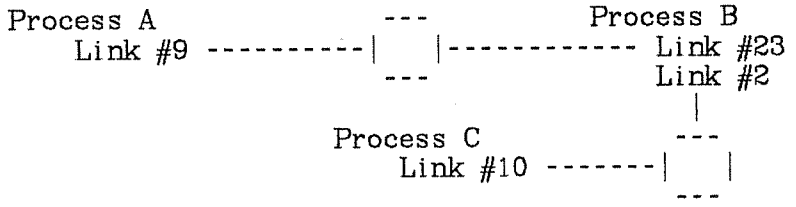
Charlotte provides a unique form of inter-process communication that combines the addressing features of capabilities and the transmission facilities of message-based communication. We will discuss these aspects separately. The kernel calls associated with inter-process communication are summarized at the end of this section.

2.1. Addressing

A *link* is a connection between two processes, each of which has a capability to one end of the link. Processes therefore address each other by presenting these capa-

bilities, which are represented by local names called *link identifiers*. Link identifiers are assigned by the kernel when the link is formed. (When we speak of "the kernel", we include the case in which the two ends are on separate machines, each governed by its own kernel.) Information about the name of and route to the process at the other end of the link is stored by the kernel for both ends.

The following picture shows three processes and two links. Links are represented by boxes.

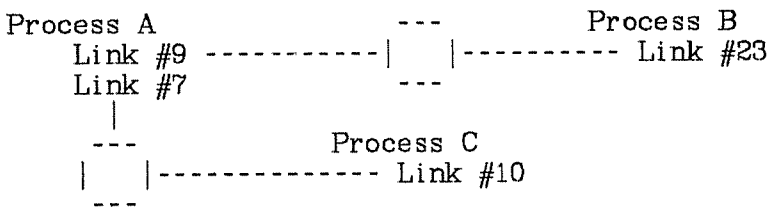


Process A uses the link identifier 9 for the same link that process B calls link 23.

Every kernel maintains a link table on behalf of all the processes on that machine. The link identifiers used by the processes are treated as indices into the table by the kernel. Each entry in the table indicates the machine, process, and link number of both the local and the remote end of the link. Information about both ends is enclosed in each message sent on the link. The receiving kernel uses this information to associate the message with a particular process and link.

2.1.1. Enclosed links

A process may give its end of a link away by enclosing it in a message. If process B gives away its link 2 in a message sent on link 23, the result is:



Process A not only receives a message from B, it also gets a new link, to which we have arbitrarily assigned link identifier 7. Process C is not aware that the other end of its link 10 has moved. Therefore, the meaning of link 10 has changed for C, and A has acquired a new link.

2.1.2. Mechanisms

We will briefly discuss the implementation of link enclosure. We ignore details of message passing, which are discussed later. For clarity, we will distinguish between processes, like A and B, and their kernels, K(A) and K(B).

Assume that B sends its link 2 (which connects with C) to A along link 23, as described above. K(B) describes this link in a message to K(A) and marks link 2 as "in transit". K(A) tentatively installs the link in its link table, say as link 7. K(A) sends an *update* message along the new link, number 7, to inform K(C) about the new location of the moved end of the link. K(C) uses this information to update its tables and sends an acknowledgement back to K(A). K(A) then marks the new link (number 7) as truly present and informs K(B) that the enclosure was successful. When K(B) hears this assurance, it removes link 2 altogether. On the other hand, if K(A) fails to get acknowledgement from K(C), then it rejects this new link (and B's message that came with it) and tells K(B) about the failure. K(B) then it marks link 2 as "usable" again and informs B that the message failed.

A difficult case arises if B and C are connected by a link, as before, but at the same time B tries to give this link to A, C tries to give its end of the link to D. The *update* messages sent by K(A) and K(D) are treated differently by C and B, respectively. The order of machine numbers C and B is used by both to allow one to accept and the other to reject to attempt to update.

2.1.3. New links

In order to introduce two colleagues A and B together, a third process C can form a link and hold both ends. C can then give one end of this link to A and the other to B. The *MakeLink* kernel call constructs a new link with the caller holding both ends.

2.1.4. Half links

One process on each machine, the kernjob, is allowed to construct links to the kernjob on any other machine. The *KernLink* kernel call provides this service. Such links are required during initialization (discussed later). Messages sent along these links are received on link 0; they cannot be used for responses from the remote end. These links are therefore called *half links*, since only one end is normal.

2.1.5. Link destruction

A process may delete a link in which it is no longer interested by submitting a *DestroyLink* request to the kernel. After this action, no new *Send* or *Receive* requests are allowed on the link. Any request that is pending is cancelled. This subject is discussed in greater detail below.

2.2. Communication

Process-level communication is

(1) non-blocking: A process can generally continue executing while the kernel is transmitting a message on its behalf,

(2) unbuffered: A message is not transmitted until the receiver has provided a place to put it, and

(3) synchronous: Processes are not interrupted by the arrival of messages.

The unit of communication is a *message*, which is a package of information of any length up to some defined maximum (about 2K bytes). Messages are assumed to have

no internal structure. A *buffer* is an area in the memory of a process that contains or is expected to receive a message. A message is sent from a *sending process* (or simply *sender*) by submitting a request to the *sending kernel*. It is received by the *receiving process* (or *receiver*) with the help of the *receiving kernel*. In order to transmit messages, kernels exchange *packets*. A packet contains zero or one messages as well as additional information. We present the sending and receiving kernels as distinct, although obvious optimizations are possible if they are the same.

2.2.1. Process view

The sender initiates a transmission with the *Send* kernel call, in which it supplies a buffer of data to the sending kernel. The receiver indicates willingness to receive a message using the *Receive* kernel call, in which it supplies a buffer to the receiving kernel. After a matching *Send* and *Receive* have been executed, the kernels cooperate to transmit the data and return the results to the processes if they are waiting for it, or store it for later *Wait*.

Completion of a *Send* or *Receive* operation is called an *event*. A process tests for an event by executing a *Wait* kernel call, which delays the process until at least one event has occurred and returns an indication of an event. A *Send* or *Receive* together with the matching completion event is called a *transaction*.

2.2.2. Specifications

All kernel calls are shown in detail at the end of Section 2.

2.2.2.1. Send

The *Send* kernel call specifies a link identifier and a buffer. (It also specifies an enclosed link, if any, but we will ignore this aspect of communications here.) It returns either the link identifier or a negative error code. (The return value, whichever it is, may be used immediately as an argument to *Wait*, described below.) Send

initiates a transfer of data from the indicated buffer along the indicated link. The send transaction remains in progress until its completion event is indicated by the *Wait* call (see below). It is ill-advised to change the contents of the buffer before the transaction completes. It is an error to issue a send over a link on which a *Send* is still in progress.

Errors are indicated as part of the completion event. Some errors (such as invalid arguments) cause an immediate completion. The value returned from *Send* will be an error code. Others (such as remote process dead) may not be signaled for a while. The errors include use of a destroyed link, enclosing a destroyed link, and enclosing a link on which a *Send* is outstanding.

2.2.2.2. Receive

The *Receive* call specifies a link and a buffer. The link may be the special form *ALL_LINKS*. This call also returns the link number or an error number. Receive allows a message to be received on the indicated link (or any link) and placed in the buffer. If the buffer is not large enough to hold the entire message, only the head of the message is placed in the buffer, the tail is lost, and the completion events for both the *Receive* and the matching *Send* transactions indicate the fact that fewer bytes than expected were transferred.

The receive remains pending until its completion event is indicated by the *Wait* call (see below). It is ill-advised to read or modify the buffer until the transaction completes. It is an error if the indicated link currently has a *Receive* pending. *Receive* on *ALL_LINKS* precludes any simultaneous *Receive* on individual links, including *Receive* calls that have completed but have not yet been reported to the process through a *Wait* call. Errors (including errors in the syntax or arguments of the *Receive* call) are indicated in the completion event.

If a *Receive* on All_LINKS is matched by several *Send*s (which may have been issued long before the *Receive* was requested), only one of them will get through and the others will remain pending. The *Receive* is satisfied by the arrival of one message. The *Send* selected will not necessarily be the first one signaled to the receiving kernel (see **Implementation** below).

It is illegal to have more than one *Receive* pending on the same link. In particular, a *Receive* on ALL_LINKS must not be pending at the same time as any other *Receive*.

2.2.2.3. Wait

The *Wait* call delays the calling process until an event occurs. The caller may either specify the link on which the event is expected, in which case *Wait* only returns when this event happens, or may specify ALL_LINKS, in which case an event on any link is reported. In addition, the caller indicates whether incoming, outgoing, or both kinds of messages are to be signaled. For example, it is possible to await the completion of an outgoing message (that is, a *Send*) on link #7. The event returned is a combination of a link identifier, a completion code, direction (Send or Receive), and an enclosed link number, if any.

If the link is one of the error codes returned by *Send* or *Receive*, then *Wait* returns an error. This feature allows *Wait* to take *Send* or *Receive* as its argument. If there is no transaction pending in the given direction on the link (all transactions that have been started have already been signalled through other calls to *Wait*), then *Wait* returns an error.

If an event has already occurred when *Wait* is called, the call returns immediately (subject to scheduling delays). If more than one event is pending, only one is indicated. The completion code is an enumeration type, one of whose values is *Success*. Other values indicate abnormal situations such as invalid arguments on the

corresponding *Send* or *Receive*, a *Receive* with too small a buffer, a destroyed link, or a broken kernel. In addition, the event indicates the number of bytes received by a *Receive* call.

2.2.2.4. Cancel

The *Cancel* call takes a link number and a direction. A *Receive* on ALL_LINKS is *Cancel*led by specifying ALL_LINKS as the link number. The direction specifies that the *Send* or *Receive* transaction is to be cancelled on that link if possible. *Cancel* returns either success or failure. Failure occurs if the operation has progressed beyond the point where the kernel can stop it. This call blocks the process until the kernel is able to report success or failure.

If there is a *Receive* on ALL_LINKS pending, and a *Send* on some link is *Cancel*led, then that latter link is added to the list of links to which the *Receive* applies. If a *Send* with an enclosed link is cancelled before a *Receive* on ALL_LINKS completes, the previously enclosed link is added to the list of links to which the *Receive* applies.

2.2.2.5. Destroy

The *Destroy* call destroys a link. This call is legal even if a *Send* or *Receive* is pending on that link. No notification will be provided of the success or failure of such operations if the link is destroyed. If a *Send* or *Receive* had already completed, but the event had not been noticed yet, because no *Wait* had been submitted, the event is lost. *Destroy* always succeeds, but it blocks the process until the kernel can complete its work. Further *Send* and *Receive* operations on this link are not allowed.

As a side-effect of destroying a link, the kernel causes a message to be sent on that link to the remote end, indicating that the link has been destroyed. This message causes any *Send* or *Receive* pending on the remote end to terminate with completion code "link destroyed". The process at the remote end must explicitly *Destroy* its end

of the link as well before the kernel will discard it; until then, all operations on the link fail with the completion code "link destroyed". As a special case, a link that is being used to *Send* or *Receive* an enclosed link cannot be destroyed until that *Send* or *Receive* completes. A process attempting to destroy the link will block until then.

2.2.3. Discussion

We pause here to discuss some of the design decisions in the above specifications.

2.2.3.1. Selective Receive

We restrict *Receive* to specify a single link or *ALL_LINKS*. Alternatively, we could allow *Receive* to specify a set of links. The process may consider a set of links to be equivalent in the sense that messages arriving on any of them are expected to need the same size buffer and require the same sort of processing. A receiver can economize on buffer space by tying one buffer to all of the links and processing incoming messages from any of the links first-come-first-served. However, this buffer is in danger of being overwritten by receipt of new messages while an old one is being processed.

However, such intermediate levels of selectivity are less important here than in other proposals. A separate *Receive* request may be started for each link of interest, and one *Wait* will report the one that succeeds. To avoid selecting a particular link, it is enough to avoid submitting a *Receive* request for it.

2.2.3.2. Event Notification

Processes can be notified of events in several alternative ways. The concept of events as specified above allows asynchronous notification to be added as a fairly natural extension: A procedure could be bound to a link identifier. When a transaction on that link completes, the procedure is called with arguments much like those returned from *Wait*.

It might also be useful to allow a process to test for a given event without waiting for it. To make such a facility truly useful, however, there also has to be a way for the process to remove the event from the queue of pending events, either by the convention that a "true" return from a "test event" call removes the pending status, or by another kernel call to remove that status explicitly. More generally, we could provide all sorts of facilities for examining and rearranging the queue of pending events.

The *Wait* kernel call waits indefinitely for an event to occur. The opposite extreme (wait no time) reduces to the "test event" call discussed above. The problem with a "wait with timeout" call is that a simple-minded process has no idea what a reasonable timeout should be. Timeouts are not necessary to guard against being blocked by a *Send* to a dead process, since such requests terminate with an error indication. If the target is not dead but looping, the target is simply reflecting that "catatonic" behavior back to the requester. Ultimately, there is some sophisticated process (possibly a human waiting at a terminal) that loses patience and kills the entire chain.

A sophisticated process, on the other hand, can use a link to an "alarm clock" server. It sends a timeout request over the link and posts a receive on it. An ordinary blocking *Wait* on *ALL_LINKS* is then guaranteed to return after a fixed maximum limit.

2.2.3.3. Receive on ALL_LINKS

Receive on *ALL_LINKS* precludes any simultaneous *Receive* on individual links. We have not encountered situations where we need both a general and a specific receive at the same time. The implementation of *Receive* becomes complicated if we allow both at once. First, a message that has been received on All_LINKS but not yet discovered by the process via *Wait* must be copied into another buffer if the process subsequently tries to *Receive* on the link over which the message arrived. Second, if a specific *Receive* is issued in the middle of negotiations between kernels concerning a

message that satisfies a general *Receive*, decisions have to be changed. Third, it can be clumsy for the process to discover which of two *Receive* requests completed so that a new *Receive* request can be generated. For these reasons, we have chosen to prohibit such simultaneous requests.

2.2.3.4. Multiple actions

A second *Send* or *Receive* on a link could be treated as a *Wait* followed by the *Send* or *Receive*. This interpretation reduces the number of kernel calls necessary in many cases. A process that knows a *Send* has completed because a matching *Receive* has finished need not *Wait* for that *Send*. A producer can alternate send buffers without fear of overwriting:

```
loop
        fill buffer 1 (* buffer 1 is certainly not undergoing Send *)
        send buffer 1
        fill buffer 2 (* buffer 2 is certainly not undergoing Send *)
        send buffer 2
endloop
```

A consumer can use similar code.

2.2.3.5. Transaction identifiers

Sending two messages simultaneously over the same link would cause no problem of disambiguation if each transaction were given a unique identifier. We have decided not to use this technique, however, since it is not clear how important such uses would be, nor whether they are worth the extra kernel cost of managing transaction numbers.

2.2.3.6. Error Events

Some errors, such as communications failures or dead processes can only be signaled to a process after a delay and are thus best indicated as a particular kind of "completion event". Other errors, such as invalid arguments to *Send* or *Receive*, are

indicated immediately by returning a special link identifier. If the process wishes, this number can be checked immediately. If not, the process will discover the failure the first time it executes *Wait*.

Providing a receive buffer that is too small to accept an incoming message is not necessarily a programming error. Both the sender and the receiver will be notified of this situation as part of the completion event. It is the responsibility of higher level protocols to use this information. A fixed but generous upper bound on message size allows naive processes to avoid worrying about this problem. On the other hand, this is another argument in favor of allowing *Receive* to specify a set of links, since a process may be expecting small messages on some links and large ones on others.

When a link is destroyed, then all transactions (whether to *Receive* or *Send*) on both sides of the link are aborted, with notification in the completion event. On the other hand, transactions that have already completed but have not yet been awaited are unaffected. The process that did not instigate the link destruction is notified either by a Wait that reports an aborted transaction or by trying a new transaction and discovering then that the link has been destroyed. The link is not destroyed on this end until the process explicitly destroys it.

2.3. Implementation

The interprocess communication described above is implemented by means of a finite-state automaton for each link. These automata are packaged together in one program that accepts packets for any link and modifies the state of the appropriate automaton, possibly emitting packets as well. A message sent between two processes generates a number of such packets. It is assumed that the underlying nugget delivers these packets reliably.

In order to explain the algorithm used for communication, we will present some increasingly complex scenarios.

The following abbreviations will be used in describing packets:

L	the link involved
M	the enclosed (Moved) link (if any)
S	the client that initiates *Send*
R	the client that performs *Receive*
C	the client at the other end of the enclosed link M

The following packet types are used:

REQ(L,S,R,M) a send request on link "L" from client "S" to "R" with link "M" enclosed

ACCEPT(L,R,S) an acknowledgement to previous send request on link "L", indicating that the data have been received

REJECT(L,R,S) a negative acknowledgement to previous send request on link "L"; the enclosed link could not be moved.

NEED_DATA(L,R,S) an acknowledgement to the previous send request on link "L" asking "S" to send the data again.

DATA(L,S,R) the re-transmission of the sender after the receiver made a "NEED_DATA" request.

CANCEL(L,S,R) the sender wants to cancel the previous send request.

CANCEL_OK(L,R,S) the cancel is granted by the receiver side.

DESTROY(L,S,R) a request to destroy the link.

DESTROY_OK(L,S,R) an acknowledgement to the request to destroy.

UPDATE(M,R,C) a request to move the other end of the link to a new client.

UPDATE_OK(M,C,R) an acknowledgement to the request to move a link.

UPDATE_FAIL(M,C,R) a negative acknowledgement to the request to move a link.

RETRY(L,R,S) the receiver asks the sender to resend the data and request again after the sender is informed of the new link address, which is about to change, since the receiver is moving its end of the link.

2.3.1. Send and Receive without Link Enclosure

Case I: B's client has performed *Receive* in time.

A-->B	REQ(L,NOLINK)	A sends request and data to B
A<--B	ACCEPT(L)	B accepts the data

case II: B's client performs *Receive* after the *Send*.

A-->B	REQ(L,NOLINK)	A sends request and data to B
A<--B	NEED_DATA(L)	B needs A to resend the data again
A-->B	DATA(L)	A resends the data

2.3.2. Send and Receive with Link Enclosure

link M is between client A and client C.

case I: no outstanding send from C to A

A-->B	REQ(L,M)	A sends request, data, enclosure to B
B-->C	UPDATE(M)	B needs C's permission
B<--C	UPDATE_OK(M)	C permits B
A<--B	ACCEPT(L)	B acknowledges both data and link

case II: interfering send request from C to A on link M

A-->B	REQ(L,M)	A sends request, data, enclosure to B
A<------C	REQ(M,NOLINK)	C sends request, data to A
B-->C	UPDATE(M)	B needs C's permission
A------>C	RETRY(M)	C should retry send request latter
B<--C	UPDATE_OK(M)	C now allows update
A<--B	ACCEPT(L)	B is happy
B<--C	RQ(M,NOLINK)	C repeats request, now to B

2.3.3. Cancel an outstanding send

case I: client B has not done *Receive*, so cancel is fine.

A-->B	REQ(L)	
A-->B	CANCEL(L)	cancel the previous send request
A<--B	CANCEL_OK(L)	

case II: B has already accepted the data, so cancel fails

A-->B	REQ(L)	
A-->B	CANCEL(L)	cancel the previous send request
A<--B	ACCEPT(L)	cancel fails

case III: client B had not done a *Receive*

A-->B	REQ(L)	
A-->B	CANCEL(L)	cancel the previous send request
A<--B	NEED_DATA(L)	client B asks to resend the data
		cancel succeeds

case IV: B rejects the *Send* request, so cancel succeeds.

A-->B	REQ(L)	
A-->B	CANCEL(L)	cancel the previous send request
A<--B	REJECT(L)	

case V: B had enclosed L elsewhere, so A is told to retry.

A-->B	REQ(L)	
A-->B	CANCEL(L)	cancel the previous send request
A<--B	RETRY(L)	cancel succeeds

2.3.4. Cancel an outstanding receive

No packet is generated for canceling a receive request. The cancel will fail when a send request from the sender has arrived and the receiver has committed this transaction by either sending "accept" or "need data" to the sender's kernel. The sender may still ask to cancel the transaction even after the receiver sends out "need data" message, so cancelling a *Receive* can succeed even after the receiving kernel has committed the transaction.

2.3.5. Destroy a link

case I: no outstanding send/receive

A-->B	DESTROY(L)	A destroys link to B
A<--B	DESTROY_OK(L)	B grants the destroy request

case II: A is receiving a link from B across link L, and A has sent out "UPDATE" to C. A is now waiting for a reply from C. In this case, A waits until the receive transaction completes to issue the DESTROY request.

case III: the other end of link L is moving

B-->C	REQ(U,L)	B wants to move link L(A<-->B) to C
A-->B	DESTROY(L)	A wants to destroy link L
B-->C	CANCEL(U)	B must cancel previous send request

This CANCEL will always succeed: Even if C has a matching receive and is going to update information on A, C will get "UPDATE_FAIL" since link L is half destroyed by A. After cancel succeeds:

A<--B	DESTROY_OK(L)	allows A to destroy link.

case IV: both sides want to destroy a link

A-->B	DESTROY(L)	A wants to destroy link L
A<--B	DESTROY(L)	B wants to destroy link L also

2.4. Specifications The specification of kernel calls and associated data structures is stored in the file "syscalls.u.h" in the "src/include" directory of Charlotte. We give an excerpt of that file here. The actual declarations are subject to change; any program using these calls should include that file.

```
const
        (* Values returned from kernel calls.  Some of these are also
        used as completion codes for transactions. *)
                SUCCESS      =  0;
                FAILURE      = -1;
                UNDEF_USER   = -2; (* Specified user does not exist. *)
                UNDEF_LINK   = -3; (* Specified link does not exist. *)
                RECBUFUNFLOW = -6; (* Receive buffer too small *)
                KERNELDOWN   = -7; (* Not getting response from other kernel *)
                BADENCLOSURE = -8; (* Enclosed link involved in pending Send
                        or Receive, or not accepted by remote end (other end of the
                        link may have been in motion as well). *)

        (* Maximum size (in bytes) of the Contents field of a DATA message. *)
                MESSAGEMAX   = 2000;

        (* Predefined link numbers. *)
                NOLINK       = -1; (* Used to specify no link. *)
                ALL_LINKS    = -2; (* Used to specify all active links. *)

type
        ReturnCode  = integer;      (* Return codes from system calls. *)
        CompletionCode = integer;           (* completion code for transaction *)
        LinkId      = midint ;              (* Link numbers. *)
```

```
(*
* Events returned from the Wait call.
*)
        Direction = ( Sent, Received, All);
        Event =
                record
                        transaction   : LinkId;
                        direction     : Direction;
                        code          : CompletionCode;   (* completion code *)
                        length        : integer; (* bytes *)
                        enclosure     : LinkId;
                end;

procedure Send (
        lnk : LinkID;
        msg : integer;  (* virtual address of data buffer *)
        size : integer; (* in bytes *)
        enclosure : LinkID;
        ) : ReturnCode;

procedure Receive (
        lnk : LinkID;
        msg : integer;  (* virtual address of data buffer *)
        size : integer; (* in bytes *)
        ) : ReturnCode;

procedure Wait(
        lnk : LinkID;
        direct : Direction;
        var Result : Event
        ) : ReturnCode;

procedure LinkDestroy (
        lnk : LinkID
        ) : ReturnCode;

procedure MakeLink (
        var lnk1, lnk2 : LinkID
        );

procedure KernLink (
        machno : MachineID
        ) : LinkID;
```

```
procedure Cancel (
        lnk : LinkID
        direct : Direction;
        ) : ReturnCode;
```

---

3. Process Control

Processes under Charlotte do not share memory. However, the KernJob process has special calls available for examining and depositing in the memory of processes on its machine, and the KernJob makes these facilities available to holders of "control" links. (These facilities are detailed in the KernJob documentation below.)

The kernel uses round-robin scheduling among active processes, with a fixed quantum (currently 3/60 second). The holder of a control link can deactivate a process as part of a long-term scheduling policy.

We will first discuss the process environment, ways a process can influence this environment, and how new processes can be created.

3.1. Process environment

Charlotte provides each process with a virtual address space. Charlotte may decide to page this space (at present it does not); such treatment is invisible to processes. The virtual address space is composed of one region for text, initialized data, and uninitialized data, and a second region for the stack.

The Starter utility (described later) assumes that processes are described by *Object Modules* in the form used by Berkeley Unix Version 4.1. When the Starter loads a process, it allocates sufficient room for the combined text, data, and bss areas.

There is presently no way a process can request more space.

3.2. Termination

A process ceases execution by executing the *Terminate* kernel call. Any link owned by the process is destroyed. The KernJob on the machine where this process lived is notified and may take action, in particular, to destroy any control link for the process.

3.3. Creating processes

New processes are constructed by the KernJob through the restricted *MakeProcess* call, which returns a process identifier that can later be used to control the process. The Starter process, which manages memory, asks the KernJob on the appropriate machine to execute *MakeProcess* on its behalf. Arguments to *MakeProcess* describe the memory requirements of the new process: where it starts in physical store and how long it is.

3.4. Initialization and Recovery

The GetMemoryMap kernel call is used by the KernJob to discover the initial arrangement of memory and then is given to the Starter so it can assume its responsibility of managing this resource.

In order to provide for reconstruction after partial network failure, a KernJob can derive information for every process in the machine to give to the Starter by using the GetProcessDesc kernel call.

3.5. Specifications

```
type
        PageId = integer;    (* Physical page frame numbers.    *)
        UsrAddr = integer;
        MachineId = integer;
        ProcId = integer;


(*
 * Record type returned by the GetMemoryMap kernel call.  This
 * structure describes the physical memory available for user
 * processes.
 *)
        MemoryMap =
                record
                        MachineNumber : MachineID;
                        FreeMemStart  : PageId
                        FreeMemSize   : integer; (* blocks *)
                end;


(*
 * Record type returned by the GetProcessDesc kernel call.  This
 * structure describes the physical memory occupied by a particular process.
 *)
        ProcessDescription =
                record
                        ProcessId     : integer;
                        ControlLink   : LinkID;
                        ImageStart    : PageId
                        ImageSize     : integer;
                        StackStart    : PageId;
                        StackSize     : integer;
                end;

procedure Terminate;

procedure MakeProcess (
        imagestart, stackstart : PageId;
        imagelength, stacklength : integer;
        link: LinkId
        ) : ProcId;

procedure GetMemoryMap (
        var memmap : MemoryMap
        ) : ReturnCode;

procedure GetProcessDesc (
        lastproc : integer; (* which one we have just seen; we want the next *)
        var procdesc : ProcessDescription
        ) : ReturnCode;
```

```
procedure KernLink (machno : MachineId) : LinkId;

procedure Peek (
        PId : ProcId;
        startloc : UsrAddr;
        length, bufaddr (*address*) : integer
        ) : ReturnCode;

procedure Poke (
        PId : ProcId;
        startloc: UsrAddr;
        length, bufaddr (*address*) : integer
        ) : ReturnCode;

procedure Inspire (PId : ProcId) : ReturnCode;

procedure Expire (PId : ProcId) : ReturnCode;

procedure Suspend ( PId : ProcId) : ReturnCode;

procedure Resume ( PId : ProcId): ReturnCode;
```

---

4. I/O

Most high-level input and output are provided by utility processes. However, some very low-level facilities are provided by the Charlotte kernel. We will discuss aspects of both these levels.

4.1. Low-level I/O

The *PutChar* and *Getchar* routines are provided for low-level control of the console terminal on each machine. These calls are intended to be used by the Terminal Driver process. The *PutChar* call adds the character to a list of characters ready to be printed out; the caller is blocked only if this list has overflowed. *Getchar* gives the caller the next character, blocking it if necessary.

The kernel echoes all characters on the terminal. If characters are typed in faster than the program is accepting them, characters that are lost are echoed by ^G (bell).

4.1.1. Specifications

---

procedure PutChar (ch : char);

procedure GetChar : char;

4.2. High-level I/O

A frequent situation that is represented by a pipe in Unix has a producer creating information that is directed to a consumer accepting this information. These two processes may have bursty behavior. Occasionally the consumer may wish to modify the behavior of the producer.

For example, the producer may be a file server for a file opened for reading. The consumer is a process reading the file. The occasional modifications are requests to seek to another region of the file.

In another example, the producer is the Terminal Server producing lines of data for a process reading from the terminal. Occasionally the consumer may wish to set echoing characteristics.

The producer may be a process creating data, and the consumer may be a file server accepting this data into a file opened for writing. In this case, there are no messages in the reverse direction.

Interposed between the producer and the consumer may be a Buffer process, whose duty is to even out the bursty nature of its two clients. This Buffer should appear to the consumer to be a producer, and should appear to the producer to be a

consumer. Reverse-direction messages from the consumer to the producer should flow through the Buffer.

The semantics of *Send* and *Receive* were designed for uniform treatment of all these producer-consumer cases. The convention that Charlotte processes are expected (but not forced) to obey is the following: Given a link to a producer, the consumer may *Receive* whenever it wishes more data. It can then *Wait* until the data arrive, or it may perform other duties in the meantime. There is no need to request the next batch of data explicitly. Given a link to a consumer, the producer may *Send* whenever it has more data and the previous data have been *Received.*

The producer should also have a *Receive* pending on the same link so it can detect requests from the consumer. These requests are always composed of two messages. The first alerts the producer that a request is coming. The producer should cancel the current *Send,* if there is one, and then *Receive* the second part of the request. The request is then serviced, and the result is transmitted to the consumer via a *Send.* Then the normal situation resumes, in which the producer continues to *Send* as fast as it can and the consumer is willing to *Receive.*

In some cases, it is not necessary to respond explicitly to the request from a consumer. For example, if the consumer wishes to seek randomly in a file, it is not necessary to respond, only to properly send the next section of data from the new point in the file. In such cases, the first *Send* after receipt of the second half of the request is a normal data *Send.*

In other cases, the consumer has asked a question that must be answered, like "where is the current file pointer?" or "what is the current mode on the terminal?". In these cases, the first *Send* after receipt of the second half of the request answers the question, and further *Sends* revert to data.

If the producer and the consumer have not agreed on the proper amount of data to be transferred with each message, the producer might send more than the consumer is ready to receive. In this case, both parties are informed that the transmission succeeded and how many bytes were transmitted. They can use this information to adjust their buffer sizes.

5. Utilities

5.1. The KernJob

5.1.1. Introduction

One of the design decisions for the Charlotte operating system is to keep the kernel, which will reside on the each node machine, efficient, concise, and easily implemented. As a result, only those services essential to the entire system are included in the kernel, such as inter-process communication and process control. All other services are implemented through utility processes, which wait for requests coming from client links.

The KernJob is a utility process always resident on every node. It acts a representative on this node for programs that need special actions. In particular, the Starter process controlling this node may reside on a different node. It uses the KernJob's ability to make and control processes. Control links governing these created processes are implemented by links to the KernJob and by kernel calls only allowed to the KernJob.

5.1.2. KernJob Services

The KernJob provides services for process control, including:

1. To get the memory allocation map of the node machine: GETMEMORYMAP. This information is needed by the Starter during initialization and during

failure recovery.

2. To get the existent process descriptions: GETPROCESSDESC. This information is also needed by the Starter during initialization and during failure recovery.
3. To make a new process: MAKEPROCESS.
4. To exercise control over the created process: PEEK, POKE, INSPIRE, EXPIRE, STATUS, SUSPEND, and RESUME.

All these services are accomplished through the privileged kernel calls allowed only for the KernJob. The KernJob maintains a table that associates control links with processes.

The main client of the KernJob is the Starter, which asks the KernJob to create new processes on behalf of its own clients. If the request succeeds, the new control link is returned to the Starter. The Starter may, of course, transfer the link to any other process.

The KernJob has a special "input link", predefined as link 0, that has several unusual properties. First, it may only be used for *Receive*, not *Send*. Second, messages from the kernel arrive on the input link when processes terminate. Third, messages between KernJobs arrive on the input link. A KernJob that wishes to send a note to another KernJob can use the kernel call "KernLink" to generate a link whose remote end is the input link of the specified KernJob. This facility is used during initialization and recovery.

5.1.3. Using the KernJob

In order to use the KernJob, a client program should include the file "kernjob.h" from the Charlotte include directory. This file contains the message format used to communicate with the KernJob, as well as a number of useful constant declarations.

Messages to the KernJob follow this record format:

```
KJMesg =
        record
                Action     : shortint;
                ImageStart : PageId;
                ImageSize  : integer;
                StackStart : PageId;
                StackSize  : integer;
                Msg        : array 0 : MESSAGEMAX-1 of shortint;
        end;
```

This message format is used both for receiving and sending messages between the KernJob and the client. The Action field is used by a client process to denote the particular service it requires. The actions include MAKEPROCESS, GETMEMORYMAP, GETPROCESSDESC, FORWARD, PEEK, POKE, INSPIRE, EXPIRE, SUSPEND, RESUME, STATUS, DEATH_NOTICE and NEWKERN. The Msg field is mainly used to hold the data in PEEK and POKE requests. The ImageStart, ImageSize, StackStart, and StackSize fields are used in GETMEMORYMAP, GETPROCESSDESC, MAKEPROCESS, PEEK and POKE requests to specify the corresponding memory address and the size. Any links enclosed in MAKEPROCESS and POKE are given to the affected process. Links enclosed in other requests are discarded. Not only virtual space, but also registers may be inspected and modified by PEEK and POKE. By convention, negative addresses refer to registers. Requests that are out of the bounds of the affected process cause failure returns.

The KernJob answers every request it receives with a return message in the same format as shown above. In this case, the Action field is used by the KernJob to return SUCCESS or FAILURE. With the failure message, a error code is also returned in the ImageSize field. The error codes include:

```
KJ_UNKNOWN_REQUEST   : Unknown request to KernJob.
KJ_ILLEGAL_LINK_USE  : Illegal request on the link.
KJ_NO_PROC_ROOM      : Process or link table overflow.
KJ_ILLEGAL_ACTION    : Illegal request to the process.
KJ_SYSCALL_FAIL      : kernel call failure.
```

5.1.4. Miscellaneous

During initialization, only the KernJob and the primordial connector are present on every machine, and there is a link (called INITIALLINK) between them. The primordial connector acts as a starter until the real Starter can be brought up; then the primordial connector terminates.

If the KernJob should discover that its Starter is inaccessible (either because it terminated or because its machine has failed or the network has failed), it tries to find another Starter to take over control of this node. To find a new starter, the KernJob generates half links to other KernJobs until it finds one with a working Starter, with which it then links itself. The Starter squad may, of course, decide to adjust load by giving this link to some other (surviving) Starter.

5.2. The Starter

5.2.1. Introduction

The Starter is an utility process squad that manages the creation of the new child processes for the clients. Not every node needs a Starter, since one Starter may control more than one node. To start a process, the client must have a link to a Starter. Such links can be obtained from the Switchboard. The clients send the request to the Starter with a file name from which a new process is to be created.

A client's Starter link will be passed around the squad of Starters in the course of finding a good place to create a new process. The link will remain connected to the Starter that finally performs the process creation or to the Starter that decides that the requested process cannot be created.

5.2.2. Starter Services

The major service that the Starter provides is to start a new child process under the request of a parent client. Clients send messages on the links to the Starter with a file name that corresponds to an object file (in Unix a.out format). The client may also ask for a control link for the created child process. If the Starter succeeds in starting the child, a control link will be returned to the client. The control link allows a parent to exercise some degree of control over its child.

To provide the services to the clients, the Starter is registered on the Switchboard. Therefore a client can get a link to a Starter from the *Locate* request to the Switchboard. (See the Switchboard documentation.)

The Starter maintains information about all the nodes it serves, including current memory allocation and current processes states on each node. Based on this information, the Starter can decide where in memory to place new processes (and on which node within its domain). The Starter communicates with the Fileserver to obtain the text and data for the child and with the KernJob on the appropriate machine to cause the child to start and to have the proper contents.

In most cases, a single Starter will manage more than one node machine. Each Starter also has a link to one or more other members of the Starter squad. The Starters periodically exchange information about load situations so that requests to start a new process can be directed to the most appropriate Starter.

The Starter maintains tables of information about processes for each node, the process counts for all the node machines under direct control, and the load status of all of its neighbors.

5.2.3. Using the Starter

In order to use the Starter, a client program should include the file "starter.h" from the Charlotte include directory. This file contains the message format used to communicate with the Starter as well as a number of useful constant declarations.

Messages to the Starter should be in the following record format:

```
ST_NAME_SIZE = 50;
STName = array 0:ST_NAME_SIZE-1 of char;
STMesg =
        record
                Request : shortint;
                Rcode : integer;
                Name : STName;
        end;
```

This message format is for communications between the Starter and clients as well as between the Starter and its neighbors.

The Request field is used by the client to indicate whether it wants the control link of the new child process or not. The Starter always returns a message responding to every client request, placing either SUCCESS or FAILURE in the Request field of the return message. If the client has requested a control link, the return message will enclose it. Failure returns are amplified by an error code in the Rcode field. The Name field should be a full path name of a file in load-image format.

In addition to a control link, the client may establish a communication link to the child by enclosing one end of the link in the request to create the child.

After the child process has started and the Starter gives away or destroys the control link for that process, the Starter does not to exercise any control over the process. When the child terminates, the Starter is informed by the KernJob and updates its tables.

The error codes returned by the Starter are:

ST_STARTER_ERROR:	Processing error in Starter
ST_CANT_OPEN_FILE:	The file can't be opened
ST_CANT_READ_FILE:	The file can't be read
ST_BAD_FILE_FORMAT:	The file format is wrong
ST_OUT_OF_SPACE:	No room on any node
ST_REPEATED_REQUEST:	Redundant start requests
ST_KernJob_ERROR:	Error in the KernJob
ST_FILESERVER_ERROR:	Error in the fileserver

5.2.4. Initialization and recovery

At the start of initialization, only the KernJob and the Primordial Connector exist on each machine. Other utility processes (Starter, Switchboard, Fileserver, and Connector) are loaded on appropriate machines by the Primordial Connector. Initially, the Starter is linked to a number of KernJobs, a Switchboard and a Fileserver. The Starter registers itself with the Switchboard and begins to accept requests from clients.

A KernJob that has lost contact with its Starter will find another KernJob that has a working Starter; the orphaned KernJob is then introduced to the working Starter.

5.3. The Switchboard

The Switchboard is a utility process designed to allow other processes to exchange links. It allows a client process to *register* a link under a given character string name, and it allows a process to *locate* a link registered under a given name.

These requests will be described in more detail later. Several Switchboards may be active at a time, in which case they cooperate to satisfy client requests.

5.3.1. Using the Switchboard

In order to use the Switchboard, a client program should include the file "switchboard.h" from the Charlotte include directory. This file contains the message format used to communicate with the Switchboard, as well as a number of useful constant declarations.

Messages to the Switchboard should be in the following record format:

```
SBMesg =
        record
            Request : integer;
            Arg : integer;
            Name : array 1:SB_MAX_NAME_SIZE of char;
            SearchLen : integer;
            SearchHist : array 1:SB_MAX_SEARCH_SIZE of integer;
        end;
```

The include file defines several additional fields, but these are used only for communication among Switchboards and should be ignored by clients. The Request field is used by a client process to name the particular service it requires. The Arg field is used to carry auxiliary arguments. Its use will be described in more detail later. The Name field is used to hold the character string name that is to be registered or located. The SearchLen and SearchHist fields are used to specify the maximum searching length and return the list of the searched Switchboards.

The Switchboard answers every request it receives with a return message in the format given above. The Request field is set to one of the constants SB_SUCCESS or SB_FAILURE defined in switchboard.h. If the request fails, the Arg field contains an error code and the Name field contains an error message.

A name to be registered or located is a set of keywords. Duplicate keywords are ignored, and the order in which keywords appear is immaterial. However, the first keyword is used as a heuristic to guide inter-Switchboard searches, so it is a good idea to establish conventions in which the first keyword is a generic description of the service provided, and the other keywords are modifiers. The keywords are packed in the Name field of an SBMesg separated by NEWLINE characters and terminated with a NULL. Keywords may include any characters other than NEWLINE and NULL. The total length of a name, including separators and terminator, must not exceed the constant SB_MAX_NAME_SIZE defined in switchboard.h.

5.3.2. The Register Request

Register requests are used by clients to register links with the Switchboard. The link to be registered should be enclosed with the message. Once a link has been registered, it may not be used for future requests. Register requests are indicated by the constant SB_REGISTER in the Request field of a message to the Switchboard from the client. The link enclosed with the request is then registered under the name contained in the Name field of the request message.

Links can be registered in either of two modes: *once-only* or *sharable*. A link registered in the once-only mode is given away to the first client to request it, and all record of the registration is then removed. A link registered in sharable mode always remains registered and will not be given away. Whenever a client requests such a link, a new link is made. One end of the new link is sent along the registered link to the registered process and the other end to the requesting process. The Arg field of the request message gives the registration mode, which must be one of the constants SB_ONCE or SB_SHARE defined in switchboard.h. If Arg is not a legal registration mode, SB_ONCE is assumed.

A Switchboard might not have room in its tables to satisfy a particular registration request. In that case, the request and the enclosed link are passed around the network of Switchboards until one is found that can handle the request, or until the number of visited Switchboards equals the constant SB_MAX_SEARCH_SIZE defined in switchboard.h. If no Switchboard can register the link, then the enclosed link will be destroyed, which can be detected by the registering process. If some Switchboard can satisfy the request, then the link is registered at that Switchboard and a success message is returned on the registered link. The Arg field of the return message is set to the number of Switchboards that were visited during the registration attempt, and the Name field is set to the name under which the link is registered. This name will be identical to the name in the original request, unless the original name was too long or included duplicate keywords.

5.3.3. The Locate Request

Locate messages are used by clients to get a link registered under a given name. Locate requests are indicated by the constant SB_LOCATE in the Request field of a message to the Switchboard from the client. The name to be located is contained in the Name field of the request message. If the request cannot be satisfied locally, then a depth-first search is instituted through the network of Switchboards. The link on which the request arrived is enclosed with the search. The Arg field of the request message gives an upper bound on the total number of Switchboards that may be visited in such a search, including the search originator. The maximum length of such a search is given by the constant SB_MAX_SEARCH_SIZE.

When the requested link is found, it (or a new link, if it is registered as sharable) is returned to the requester in a success message. The Arg field of the return message is set to the number of Switchboards that were visited in order to satisfy the request, and the Name field is set to name under which the requested link was registered. If

the link cannot be found by visiting the allowed number of Switchboards, then a failure message is returned. The link on which the original request arrived remains attached to the Switchboard that eventually answers the request, either positively or negatively.

For a registered name to satisfy a locate request, the keywords in the request must form a subset of the set of keywords in the registered name. The order of the keywords in the two names is unimportant. If several registered names satisfy the request, then ties are broken in favor of names that have fewer extra fields. If there are still ties, they are broken in favor of links that have been given away fewer times. Any remaining ties are broken arbitrarily.

These rules are applied at the first Switchboard through which the request passes that has any match at all to the query. A "better" match at some other Switchboard will not be found.

5.3.4. Miscellaneous

Any link to a Switchboard may be duplicated by making a new link and sending one end of the new link to the Switchboard. If the original was a registered link, the new link is not registered, but may be used to make requests to the Switchboard. Similarly, any link may be destroyed. Destroying a registered link causes the link to become unregistered. In fact, this is the only way a process can unregister a link it has registered.

The following error codes are defined in switchboard.h. They are used in the Arg field of failure messages.

```
SB_UNKNOWN_REQUEST       : Unknown Switchboard request.
SB_ILLEGAL_LINK_USE      : Illegal request on a registered link.
SB_NO_REGISTRATION_ROOM  : No room for registration.
SB_SERVICE_NOT_FOUND     : Requested service not found.
SB_SEARCH_LENGTH_EXCEEDED : Maximum search length exceeded.
```

5.3.5. Discussion

The specifications given here are a first, simple solution to the general question of what sort of registration values should be supported and how pattern matching works. Currently, a registration value is a set of fields. Pattern matching returns the value with the greatest number of matching fields. If there are none, then the request is forwarded.

This algorithm has some drawbacks:

(1) A poor match at the first switchboard is preferred to a good match elsewhere.

(2) If the match is unacceptably poor in the eyes of the client, there is no way for the client to make a more specific request.

Other techniques could be used. For example:

(1) Same as above, but also an 'exact count' on patterns that specifies how many initial fields must match. This method is fairly simple and easy to implement.

(2) Same as above, but each field has an optional 'must match' mode. This method is more general and is still not too hard to implement.

(3) Registration values are arbitrary strings, not lists of fields. Patterns are regular expressions. Quality of match may be measured in various ways:

a. Any match is fine.
b. Select match with shortest (longest) registration value.
c. Select match with greatest (fewest) number of invocations of the '*' operator.
d. Select match with greatest (fewest) number of constants matched.

The criterion might even be specified by the requester. This method is much less efficient, since all values must be inspected.

(4) Registration values are sets, not lists, of fields. Patterns are also sets. Match requires each field in the pattern to be found. This technique is fairly general and not too hard to implement. To get the effect of weak matches, queries could be made very specific at first, and if they fail, the client could then submit a weaker query. It seems likely that most clients will want requests of only one field, with exact match. Therefore, even the current method is fine.

5.4. The Fileserver

5.4.1. Introduction

The current version of the Fileserver is a simple prototype. All files reside on a Unix file system on the host machine. The Fileserver communicates with this file system through a "demon link", one end of which is connected to the Fileserver, the other end connected to a program running under Unix.

The Fileserver registers itself with the Switchboard. Any client that needs to read or write files should acquire a link to the Fileserver from the Switchboard and then communicate directly with the Fileserver. File service is provided by a squad of processes, any of which can potentially access any file. In fact, each process in the squad can only access a subset of the files directly. Any request outside its jurisdiction is transparently forwarded to the file server process that can handle it; the link to the client is passed along with the request, so the file server that has jurisdiction can then communicate with the client directly.

5.4.2. File Services

The Fileserver provides services to manipulate a file. Since Unix files are used in the implementation, the operations are identical to those available under Unix: OPEN, READ, WRITE, CREATE, and SEEK.

In order to open a file, the client sends an OPEN request with an enclosed link to the file server. The file server then turns that enclosed link into an open-file link. The open request indicates whether the file is to be open for reading or writing and what the block size for transmissions should be. When a Fileserver can't satisfy the request because of wrong permissions or non-existent file, an ERROR message is sent to the client on the link, which does not become an open-file link. If the Fileserver does succeed in opening the file, the first message is a SUCCESS message.

If the file is opened for reading, the file server begins to send data sequentially in full blocks from the file on the open-file link. As soon as the client receives one block, the next one is sent; no explicit READ call is available. If the client receives less than the amount sent by the file server, the next message will start with the unread portion of the previous message.

If the file is opened for writing, the client may send DATA messages on the open-file link. The client may try to send more than the block size, but the file server may only be willing to accept a smaller amount. DATA messages that are sent to files not open for writing are discarded.

The client may SEEK to any location in the file by sending a SEEK message to the file server. The client then sends one additional dummy message to the file server to allow the file server an opportunity to cancel the next outstanding message before the client reads the next data block.

To close the file, the client destroys the link.

An OPEN request is illegal if a file is already open on the link. A WRITE or SEEK is illegal if the link is not an open-file link. If any request includes a new link, then the new link becomes a service link to the file server, not an open-file link. The file server may choose to reject this link.

5.4.3. Proposed Implementation

A native Charlotte file server will look a bit less like Unix. File distribution will be accomplished as follows. Each file has a unique full path name. To find which fileserver has a file /a/b/c, a standard hash function will be applied to the prefix name /a/b. Extendible hashing is used to allow new disks to be brought into the network or to take old ones out. There is a reorganization cost, of course. To bring a new disk in, about half the files on an existing disk must be brought to the new disk. Protection is performed on a file-by-file basis, not by looking at the protections in directories on the path to that file. Directories do not point to files; they just mention what files exist. As a result, to delete a file, it must be removed from the server on which it resides and must also be deleted from the directory that mentions it. The deletion from the directory should be done first. Directories cannot be renamed.

5.4.4. Client-Fileserver Interface

Messages to the file server use the following message format: (These declarations are also in fileserver.h, along with the subordinate types they use.)

```
(* File action constants *)
const
        OPEN = 1;
        WRITE = 2;
        CREAT = 3;
        UNLINK = 4;
        STAT = 5;
        READLINE = 6;
        SEEK = 7;
        LINK = 8;
        DATA = 9;

FSMesg =
        record
                FileAction : longint; (* see list above *)
                Remnant : array 1:MESSAGEMAX-4 of char;
        end;
```

For DATA messages, the entire Remnant is devoted to data (if the block size requires). For OPEN messages, the Remnant is treated as having this structure:

```
FSOpenMesg =
        record
                Mode : longint;  (* as in Unix *)
                FileName : array 1:MESSAGEMAX-8 of char; (* null terminated *)
        end;
```

ERROR and SUCCESS messages do not use the Remnant at all. The SEEK request uses the Remnant like this:

```
FSSeekMesg =
        record
                Offset : longint;
                Whence : longint; (* same as in Unix *)
        end;
```

The CREAT request uses the Remnant in this way:

```
FSCreatMesg =
        record
                Mode : longint; (* same as in Unix *)
                FileName : array 1:MESSAGEMAX-8 of char; (* null terminated *)
        end;
```

5.5. The Terminal Driver

The terminal driver has not been completely designed. It will follow the producer-consumer paradigm described above.

5.6. The Connector

5.6.1. Introduction

The connector is a tool to establish initial links in a group of processes. It is implemented as a free-standing utility registered with the switchboard. A program that wishes to institute a connection episode (usually a command interpreter, but in general any top-level entity in a group of processes) will be called a "parent", and the members of the newly connected group will be called "children". Children are fresh processes called into being by the connection episode.

The program for the parent should be linked with the library routine "LinkEpisode". This routine implements the client-connector protocol. The programs for the children should each be linked with the library routine "Linkup". This routine implements the child-connector protocol.

To start an episode, the parent calls LinkEpisode, passing it a link to an open file that describes the desired group structure. The file mentions three kinds of processes, which are

1. child: a new process that should be created and loaded from backing store.
2. oldtimer: a process that already exists and is registered with the switchboard.
3. newtimer: a process that may already be registered with the switchboard, in which case it is treated as an oldtimer, or it may not be registered, in which case it must be created and loaded. It will then register itself with the switchboard so that it can become an oldtimer.

5.6.2. Structure of the connector description file The connector description file will be defined by means of BNF. The general outline of the file is to define *constants* and then to define *processes*. The *linkages* among process are described last.

5.6.3. Constants

Constants are defined to save effort later in the connector description file.

```
<constant decl> ::= "const" <const decl>
<const decl>    ::= { <identifier> "=" <expr> ";" }
```

---

example:
const N = 15; A = N+4;

Forward references are not allowed in constant declarations. Every right-hand-side operand of a constant declaration must be either a constant or an integer.

5.6.4. Processes

A process may be associated with either one or both of two names: A *switchboard name* and a *program filename*. If only the program filename is specified, the process is a child. If only the switchboard name is specified, the process is an oldtimer. If both names are specified, the process is a newtimer. In addition to simple names, the connector file may also describe *process arrays* of children.

```
<process decl>  ::= "process" <pro decl>
<pro decl>      ::= {<process ids> "=" [ <switchboard name> ]
                                                [ "@" <program filename> ] ";" }
<process ids>   ::= <identifier> [ <proc iterator> ]
<proc iterator> ::= "[" <range> "]"
<range>         ::= <expr>..<expr>
<switchboard name> ::= <identifier>
<program filename> ::= /* filename which "starter" understands*/
```

example:

```
process
        philosopher[1..N] = @/usr/crystal/philosophy/phil;
                /* process array for philosophers */
        fork[1..N] = @/usr/crystal/philosophy/fork;
                /* process array for forks */
        terminal = v-term @ /usr/utility/terminal;
                /* process terminal either registered as v-term on "switchboard"
                 or must be loaded */
```

Newtimer descriptions allow the program filename to be themselves connector description files instead of runnable files.

5.6.5. Linkup

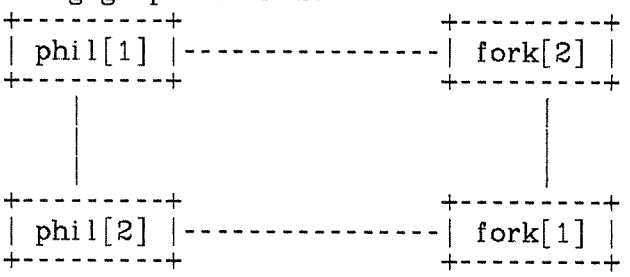
When a child starts, it should call "Linkup". This routine returns an array of links and an array of arguments. The contents of this array are described in the connector description file.

```
<linkup sec> ::= "linkup" <proc arg,links>
<proc arg,links> ::= { <processes> "=" <arguments> <links> }
<processes> ::= <identifier> [ <proc index> ]
<proc index>::= "[" <id range> "]"
<id range>  ::= <expr> | <identifier> ":" <range>
<arguments> ::= "arg" "(" <expr> { "," <expr> } ")" ","
<links>     ::= "link" "("  <link> { "," <link> }  ")"
<link>      ::= <identifier> [ <proc index> ] [ <link specifier> ]
<link specifier> ::= "(" <identifier> ")"
```

example: /* dining philosophers and their forks */

```
linkup
        phil[i:1..N] = arg(i),
                                link ( fork [ (i-1)<1? N: i-1],
                                             fork [ (i+1)>N? 1: i+1])
        fork[i:1..N] = arg(i),
                                link ( phil [ (i-1)<1? N: i-1],
                                             phil [ (i+1)>N? 1: i+1)
```

resulting graph for N=2:



A consistency check will be done to match each process's link with its peer process. A link is deemed legal if and only if both "A : link (B)" and "B: link (A)" exist.

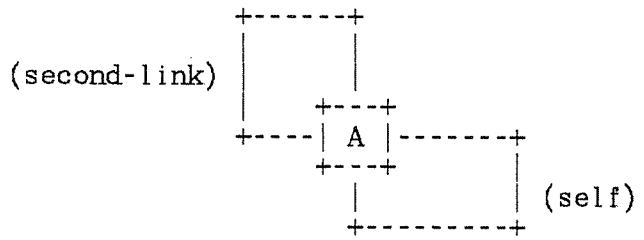
5.6.6. Link Specifier

In some special cases we allow a link specifier to specify a unique link between two processes:

1. when a process has a self link.
2. when two processes share more than one link.

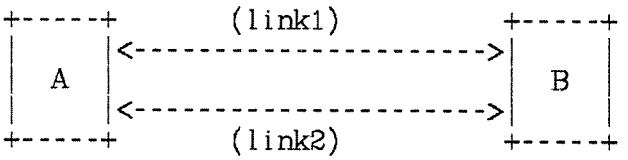
example 1:

A: link(A(self), A(self), A(second-link), A(second-link))



example 2:

A: link(B(link1), B(link2))
B: link(A(link2), A(link1))



5.6.7. Special link names

A child may wish to link with some entities such as "switchboard" and its beloved "parent". Since the parent of a child is the process that started this LinkEpisode, and is not likely to be unique as well as registered on switchboard, the name "parent" is provided to specifically name the link to it in the connector file. The switchboard is also at hand, so there is no need to search for a registration of "switchboard". Other names may be added to this list later; for now only "parent" and "switchboard" are treated specially.

example:

```
process
        child-A = @ ~crystal/foo;
        child-B = @ ~crystal/bar;
linkup
        child-A : link(parent, child-B)
        child-B : link(parent, switchboard, child-A)
```

5.7. The Command Interpreter

Part of Charlotte initialization is to build a number of command interpreter processes and attach them to terminals. The command interpreter prompts the user when ready to execute a command. The user may type the name of an executable file followed by one or more operands. The command interpreter will ask the starter to load the program and send it a link to the switchboard and an array of characters into which null-separated operands are packed. The program can receive these through the LinkUp library routine.

5.7.1. Directory Commands

The command interpreter supports relative directory addressing. It maintains a stack of directories the top of which contains the current directory. The following commands are available:

cd	change current directory
pd	push directory
p	pop directory
pw	print working directory
dir	print the stack of directories

The command interpreter prefixes the working directory to the command verb. Absolute names may be specified by beginning them with "/". Addressing of a file will be relative to the parent of the current directory if the file name is prefixed with "../". Successive levels can be backed off by adding additional "../"s to the prefix. Arguments to the directory relative commands are expanded similarly.

The first argument to any command is also expanded. This may not be desirable if commands take non-file name arguments. This feature may therefore be removed.

5.7.2. History Commands

The interpreter maintains a history of 15 commands. Any command in the history may be redone. The following commands are supported:

!!	redo last command
!<number>	redo command <number> if in history window
!<prefix>	redo the most recent command whose verb has the specified <prefix>
h	display the history window

6. Initialization and Recovery

Charlotte startup uses the same principles as we have already discussed. The startup protocol is designed to handle insertion of a new machine into a running Charlotte as well as individual machine failures. It tries not to have too much built into KernJob, which remains in existence on each machine throughout the life of the machine. The protocol uses the same load image for all machines.

Upon loading, each machine has K, the KernJob, and C, the primordial connector (not to be confused with the connector utility, discussed earlier). There is a link between K and C and a link between C and the demon D on the host machine. Each K knows its machine number. Both K and C are capable of responding on any link to the question "who are you". Each will answer "I am K (or C), on machine x". Each C first asks the K who he is, so now C knows who he is, too.

By speaking with D, each C reads the global startup file, charlotte.rc. This file uses standard Connector syntax. Here is a sample file, in simplified syntax:

```
1       D
2       A B
3       G S
4
D1 A1
A2 B1
B2 G1
S1 K[3]
S2 K[4]
```

Meaning: Machine 1 is supposed to be initialized with program D. Machine 2 has programs A and B, and so on. B has two links, the one called 1 to A, and the one called 2 to G. S has a link called 1 to the K on machine 3. The bracket syntax can be used to allow several programs to have the same name, but be distinguishable in the file.

By speaking with D and by using kernel commands to make memory and to inspire, each C creates the processes that belong on its machine. By speaking with the linkup routine, which starts each of those processes, C is able to obtain all the links that it will need. It gives away some of those links locally. In general, it gets a link from any process in the first column of the second part of the file, and gives it to the corresponding process in the second column.

K can answer the following question: "Give me a link to C[i]" for any machine i. It answers the question by forwarding a similar request to K[i] along a "half link" generated on the spot to K[i]. The request it forwards is "Give me a copy of any link you have". By talking with its K, each C gets a link to the peers it needs to talk with to distribute the remaining links. The C's distribute the remaining links. Links that are to connect K with other processes are given to K, which can handle the following request: "Here, take this link and proclaim along it the answer to 'who are you'".

Having finished this work, each C terminates. The usual result is that each K gets a link to exactly one starter program S. There may be many S's, each with several K's

under its control.

If a new machine is to be brought into the network, an entry should be made in charlotte.rc for that machine that just says "recover through machine i". When C finds that its job is to recover, it asks K "Give me a link to C[i]". The response it gets is a link to the starter controlling machine i, since that is the only link that K[i] owns at this point. C gives the link to K, saying, "Here, take this link and proclaim who you are". In this way, S discovers the presence of a new K.

If a machine fails, we assume that within some small amount of time, all kernels dealing with this machine discover the fact, and send a "link destroyed" message to all local clients. If S finds that its link to another S has disappeared, it realizes there are now two problems:

(1) The network of S programs may have been disconnected.

(2) Some machines may no longer be under control.

Disconnection of starters only results in poorer placement of new processes, since global information is no longer available. Machines that are no longer under control institute a search (described earlier under **KernJob**) for a starter that will assume control.