The Reactive Kernel

Thesis by

Jakov N. Seizovic

In Partial Fulfillment of the Requirements
for the Degree of
Master of Science

California Institute of Technology
Pasadena, California

1988

(Submitted Oct 24, 1988)
Caltech CS-TR-88-10

Acknowledgments

I would like to express my special thanks to my academic and research advisor, Chuck Seitz, for his endless patience and his guidance through the mind-boggling alternatives. My thanks also go to my fellow graduate students: Wen-King Su, Bill Athas, and Dražen Borković, for all the helpful discussions and for being there when I needed them the most; to Hal Finney of Ametek, for his many suggestions on how to improve the Reactive Kernel; and to Dian De Sha, our technical editor, who admirably carried the heavy burden of always being the first to read my drafts.

The research described in this report was sponsored in part by the Defense Advanced Research Projects Agency, DARPA Order number 6202, and monitored by the Office of Naval Research under contract number N00014-87-K-0745; and in part by grants from Intel Scientific Computers and Ametek Computer Research Division.

Contents

Chapter 1

Introduction

1.1 Background

A *multicomputer*, or *message-passing concurrent computer*, consists of N computing *nodes*, connected by a message-passing communication network (Figure 1.1).

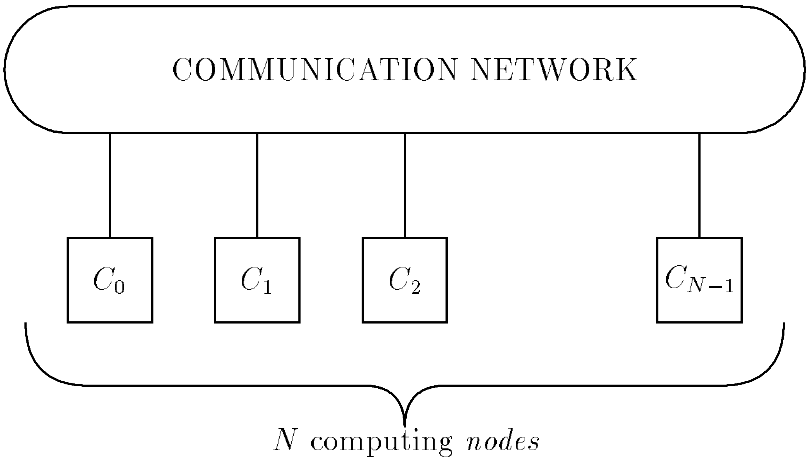

Figure 1.1: A programmer's view of a multicomputer

There is a copy of an operating system in each computing node. The node operating system supports multiple processes and provides them with an interface to the communication network (Figure 1.2). The communication network performs the message routing, enabling each process to communicate with any other process.

The development of *first-generation* multicomputers began with the Cosmic Cube project [Seitz 85], and continued with its commercial descendents, made by Intel, Ametek, and N-Cube. All of the first-generation multicomputers used a binary n-cube interconnection network and software-controlled store-and-forward message routing.

Recently, the first representatives of *second-generation* multicomputers have appeared as commercial products [Seitz *et al.* 88b]. With the advances in single-

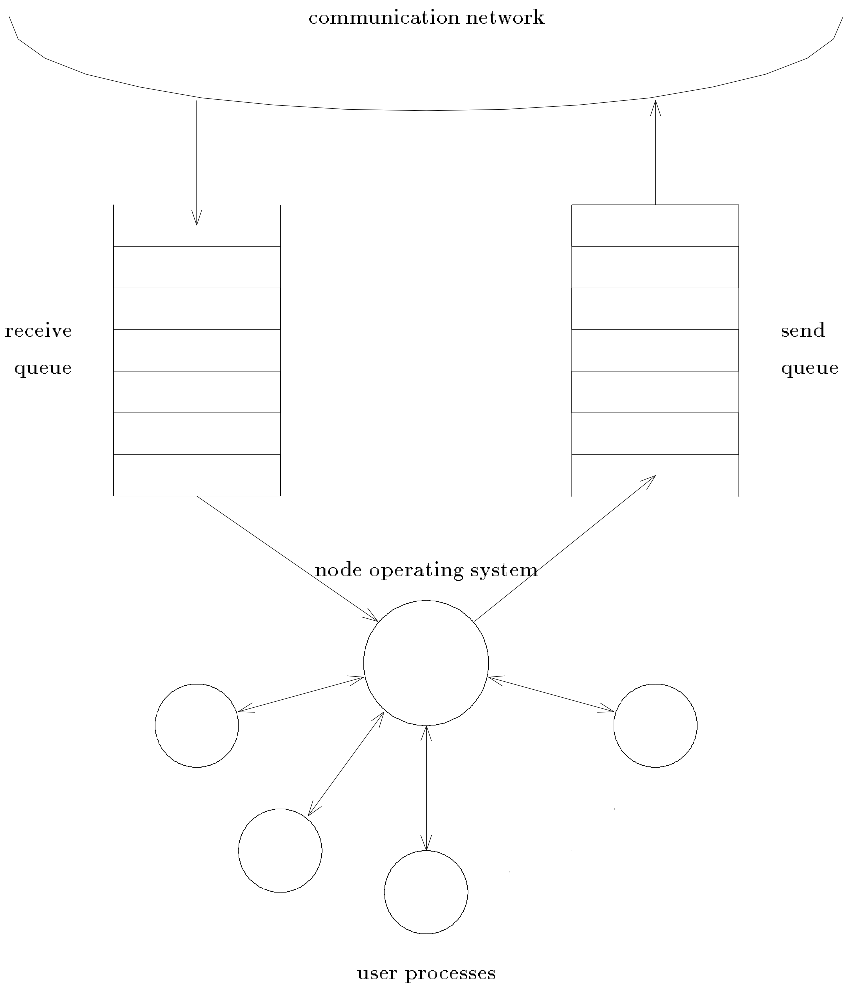

Figure 1.2: A multicomputer node

chip processor performance and RAM technology, the node performance and the memory capacity of these machines have improved by about one order of magnitude. The performance of the message-routing network, however, has improved by as much as three orders of magnitude, as a combined result of using *wormhole* routing [Dally & Seitz 87]; wide-channel and low-dimensional networks [Dally 87]; and dedicated, high-performance routing hardware [Flaig 87].

1.2 Motivation

Message Latency

As a result of the two-orders-of-magnitude improvement in the relationship between communication and computing performance of second-generation multicomputers, the network component of message latency has been reduced from a few thousands to a few tens of CPU instructions. For example, the message latency of a 40-Byte message traveling the longest distance in the 64-node hypercube network of a first-generation machine with store-and-forward routing is several milliseconds. The same message would traverse the path between two corners of an 8×8 mesh with hardware wormhole routing and a channel bandwidth of $25\frac{MB}{s}$ in $2.7\mu s$. Thus, the software component of message sending and receiving becomes the dominant part of the overall message latency.

Process Model

Programming experience with multicomputers has shown that the node operating system should not strictly enforce any particular process model. The process model that is favorable for one particular programming language or application system will bring unnecessary overhead to others. The operating system should be flexible and modular, to permit easy modifications and extensions.

1.3 The Reactive Kernel

This report describes the *Reactive Kernel* (RK), a new node operating system for multicomputers that better utilizes the performance of second-generation machines. Its main characteristics are:

- It streamlines message-handling to avoid redundant copying;
- It reduces context-switch overhead to, at most, one low-cost context switch (equal to the cost of a system function call) per message;
- Its layered structure, with well-defined interfaces between the layers, provides for easy reconfiguration and optimization depending on the programming model on hand; and
- It is a portable operating system suitable for running on both first- and second-generation multicomputers.

Most of the ideas used in the design of RK came from the extensive programming experience of W.-K. Su, C.L. Seitz, and W.C. Athas in programming the first-generation multicomputers, and were shaped into their present form during 1986. The first implementation of RK was written in 1987 for the Cosmic Cube by the author of this report. RK was later ported to the Ametek Series 2010 and Intel iPSC/II.

1.4 Overview of This Report

Chapter 2 describes the programming environment in which we use multicomputers. Chapter 3 introduces Reactive Scheduling, and establishes the conditions under which it can be used in multicomputers. Chapters 4 and 5 describe the implementation of RK, and a programming interface for processes written in C. Chapter 6 introduces a specific problem in the storage allocation in RK, and describes a solution. Chapter 7 reviews the achieved results, and discusses the possible improvements.

Chapter 2

The Programming Environment

The programming environment described here in outline form is described in detail in *The C Programmer's Abbreviated Guide to Multicomputer Programming* [Seitz *et al.* 88a].

2.1 Processes

The computation is expressed as the set of *processes*. A process is an instance of a sequential program that can include statements that cause messages to be sent and received. Each process has its own address space, and can interact with other processes only by message passing; there is no global address space.

Processes can be created dynamically during the computation, but are bound statically to the node in which they are created.

2.2 The Cosmic Environment

The *Cosmic Environment* (CE) is a host runtime system that supports the message-passing programming environment. It can handle multiple processes on network hosts, and it interfaces to one or more multicomputers.

CE in the host systems, together with RK in the multicomputer nodes, provides uniform communication between processes, independent of the multicomputer node or network host on which the processes are located. Under the CE/RK system, message-passing programs run on multicomputers, as well as across networks of workstations, on sequential computers, and on shared-memory multiprocessors. Assuming that the computation is deterministic and that it does not exceed the available computing resources, the results of the computation will not depend on the way in which processes are distributed in the entire CE/RK system.

A process within the CE/RK environment is uniquely identified with its *reference*, *ie*, the ordered pair (`node, pid`) that represents the node number and the process number within the node, respectively. The fixed numbering of nodes, together with the unique numbering of processes within each node, establishes the global name space.

2.3 Messages

A message is the logical unit of information exchange between processes. Messages can have arbitrary length, and although they may be handled differently by the system according to their length and destination, these differences are completely invisible to the programmer.

There is a bidirectional, unbounded channel between any two processes. Communication actions are asynchronous, with messages queued as necessary in transit, so that they have arbitrary delay from sender to receiver. The message order is preserved between two processes in direct communication.

2.4 Communication Primitives

The communication primitives of the CE/RK system have their predecessors in the communication primitives that were used in the *Cosmic Kernel* (CK) [Su *et al.* 87], which was the original operating system for the Cosmic Cube, and in the versions of CK that were written for the commercial multicomputers.

One of the differences between CK and RK is that in RK messages are sent and received from dynamically allocated memory that is accessed both by user processes and by the message system. Message buffers are character arrays with no presumed structure. Message space can be allocated by:

```
p = xmalloc (length);
```

and deallocated by:

```
xfree (p);
```

The `xmalloc` and `xfree` functions are semantically identical to the UNIX `malloc` and `free`, except that functions `xmalloc` and `xfree` operate on message space.

When message space has been allocated, and a message has been built into it, the message can be sent by:

```
xsend (p, node, pid);
```

The `xsend` function also deallocates the message space; it is just like `xfree`, except that it also sends a message.

Messages are received by:

```
p = xrecvb ();
```

and the pointer to the first available message will be returned. This is a blocking function; it does not return until a message arrives for the requesting process. The `xrecvb` function is like `xmalloc`, except that the contents and length of the received message determine the initial contents and length of the block. After the message is no longer needed, it should be freed by executing `xfree(p)` or `xsend(p,node,pid)`.

Besides these four, most often used functions, there is a multiple-send function, `xmsend(p,count,list)`, that sends the same message to `count` destinations, specified by the `list` array; another function that returns the length of the message buffer, `xlength(p)`; and a non-blocking receive function, `xrecv()`, that is a version of `xrecvb` that may return a NULL pointer if there is no message for the requesting process.

These communication primitives are the system primitives, and were chosen because:

- They avoid unnecessary copying;
- The setup overhead is reasonably small;
- The mutual exclusion problem between the user process and the message system is solved in a clean way;
- Different communication primitives can be layered on the top of them very efficiently [Seitz *et al.* 88a];
- They fit nicely into the Reactive Programming Model (Section 3.1); and
- They map readily to implementations using native tasking systems on multiprocessors [Athas & Seitz 88].

Chapter 3

Scheduling

The Cosmic Kernel operating system, developed for the Cosmic Cube, and the subsequent versions of CK, used in commercial first-generation multicomputers, all employ conventional scheduling strategy — round-robin, with a fixed execution period — to schedule processes within a multicomputer node [Seitz 85]. This particular scheduling interleaves the execution of processes, so that they appear to operate concurrently.

The communication primitives used in these first-generation machines — non-blocking send and receive functions — attempt to exploit as much concurrency as possible by allowing the user process to run simultaneously with the messages that are being sent and received. To avoid unnecessary context switching, the `flick` primitive is used, with following semantics: A process, after determining that its computation cannot proceed until a certain message has been received or sent, suggests to the operating system that because no further progress is possible, this is a good point for scheduling another process. In this way, a process specifies its *choice point*.

The behavior of *Actors* [Agha 86] and of objects in the Cantor programming language [Athas 87] suggest that messages can be treated as tokens to run.

As part of the RK operating system experiment, we have composed these two concepts — choice points and treating the message as a token to run — into a scheduling strategy that we call *reactive scheduling*. From an early point in the development of RK, it was clear that taking advantage of the very low network component of the message latency would depend on the message-handling overhead being as small as possible. Reactive scheduling reduces the context-switching overhead to, at most, one low-cost (equal to the cost of the system function call) context switch per message.

Under the reactive scheduler, processes behave much as Actors, and the Actor model is sufficient to express any computation [Agha 86]; however, when applied to an operating system used for running programs written in conventional sequential programming languages, this model demonstrates serious deficiencies.

In this Chapter we will first describe pure reactive scheduling; we will then explain a few additions that provide:

- a clean abstraction mechanism,
- a way to handle infinite computations, and
- a way to handle user-process errors.

3.1 Reactive Scheduling

3.1.1 Scheduling Strategy

With reactive scheduling on a multicomputer node, a process will be scheduled to run only when there is a message for it. The process runs as long as it is making progress (including the sending of messages); it then specifies its choice point by executing a blocking-receive system call, thus notifying the operating system that no further progress will be possible until another message is available. It is this scheduling strategy that makes the processes *reactive*.

The behavior of a process is analogous to the behavior of an Actor, or, more precisely, a *rock-bottom* Actor.

In the reactive model, there is a single receive queue in each multicomputer node, and all messages arriving at a node are appended to that node's queue. A process is in one of two possible states: *running* or *waiting* to get a message (Figure 3.1).

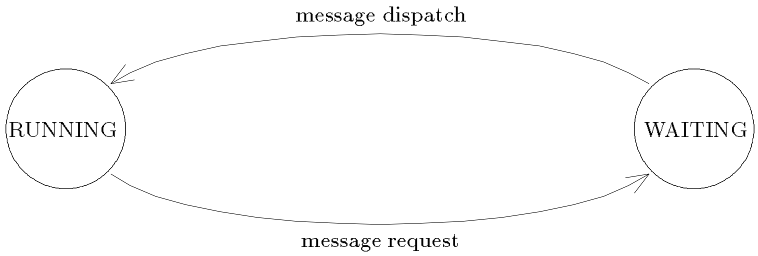

Figure 3.1: Process states

A *waiting* process, $\mathcal{P}$, starts *running* when the following two conditions have been satisfied: There are no *running* processes, and $\mathcal{P}$ is the destination of the message at the head of the receive queue.

A *running* process changes its state to *waiting* when it reaches the choice point; this is marked by the execution of the `xrecvb` system call. Therefore, when there are no *running* processes, and no messages in the receive queue, the whole node will be idle until the next message arrives. To enable a process to do some useful work even if there are no messages for it, we have modified the above strategy by allowing processes to use the non-blocking-receive system call (`xrecv`) as their choice point. Thus, when there are no *running* processes, and no messages in the receive queue, these processes are scheduled in round-robin fashion and receive a NULL message.

The `xrecvb` function can be expressed in terms of `xrecv`, but the converse is not true; therefore, in this chapter we refer to the `xrecv` as the receive system call, and, unless otherwise stated, every claim made is valid for both `xrecv` and `xrecvb`.

3.1.2 Fairness in Reactive Scheduling

The only way a process can do any useful work is by exchanging information with the outside world, *ie,* by sending and/or receiving messages. This is the most important assumption on which our choice of the scheduling strategy is based. We say that a process is *fair* if at every point of its computation it is true that it will eventually either send a message, receive a message, or terminate.

We will say that the scheduling is *weakly fair* if the following condition is satisfied: If in the receive queue there is at least one message whose destination is the process $\mathcal{P}$, and $\mathcal{P}$ is in the *waiting* state, then $\mathcal{P}$ will eventually change its state to *running*.

The necessary and sufficient condition for weak fairness in scheduling on a certain node is that each user process on that node satisfies the *reactive property*; *ie,* each *running* process eventually will either change its state to *waiting* (by executing an `xrecv`) or terminate.

If we restrict ourselves to finite computations only, all fair processes will satisfy the reactive property by definition, since they eventually terminate. However, if we are willing to support infinite computations, the producers of an infinite number of messages become an important class of processes that are fair, but that do not satisfy the reactive property. In Section 3.3, we show the simple modification of the scheduling strategy that enables us to guarantee the weak fairness in scheduling on a multicomputer node; the only requirement is that all processes on that node be fair.

3.2 The Remote Procedure Call

The reactive communication primitives defined in Section 2.4 do not enable user processes to distinguish between the messages they receive. A process executes an `xrecv` that asks for a message (whichever one happens to be first in the receive queue). Information on the 'type' of message can be made a part of the message body.

However, reactive primitives are insufficient for providing a powerful abstraction mechanism. One way of providing an abstraction mechanism is to implement the *Remote Procedure Call* (RPC), whereby a process requests a service from another process, and cannot continue the computation until the service has been completed. An implementation of the RPC using the reactive primitives could use a *reply* message that would be sent to the requester to acknowledge the completion of the requested task. The reply message would have to be distinguishable from all other messages that could arrive for the requesting process before the reply message itself. The user of the RPC would have to know the structure of the reply message and would have to make the structure of all other messages involved in the computation distinguishable from it. This is contrary to the idea behind abstraction, in which modules should be used as black boxes with certain I/O behavior.

To enable the RPC to be used as an abstraction mechanism, we have augmented the purely reactive model so that the process can be in one of three states: *running*, *waiting*, or *blocked* (Figure 3.2).

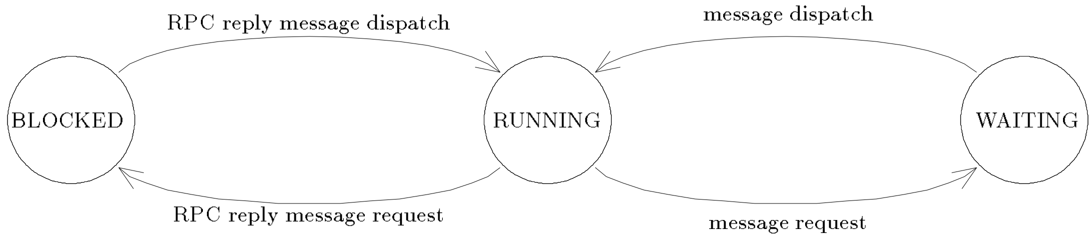

Figure 3.2: Process states

Each message has an additional field in the header to designate whether or not it is a RPC reply message. Two additional message primitives are defined: The `xsendrpc` sends the reply message, and the `xrecvrpc` puts the process into the *blocked* state, where it remains until a reply message arrives for it.

All non-reply messages that arrive for the process in the *blocked* state have to be queued for it, and delivered later. A simple change in scheduler can provide for this: Instead of always scheduling the process that is the destination of the message at the head of the receive queue, the scheduler will traverse the queue, skipping the non-reply messages destined for processes that are in the *blocked* state. If a reply message is found in the queue, and if its destination process is *waiting*, this is considered to be a programming error. Keeping the undelivered messages in the receive queue is inefficient; a description of an efficient implementation can be found in Section 5.2.1.

We will also have to change our definition of weak fairness to comply with the new process state graph. The scheduling is weakly fair if the following condition is satisfied: If in the receive queue there is a reply message whose destination is process $\mathcal{P}$, and $\mathcal{P}$ is in the *blocked* state, or if in the receive queue there is at least one non-reply message for $\mathcal{P}$, and $\mathcal{P}$ is in the *waiting* state, then $\mathcal{P}$ will eventually change its state to *running*.

If all processes on a node satisfy the reactive property, this modified version of scheduling will still be weakly fair.

A sufficient condition to guarantee that the above-described error (a reply message arriving for a *waiting* process) will not occur is that the first state change of a process that issued the RPC request be from *running* to *blocked*.

3.3 Infinite Computations

Infinite computations are legal in Actor systems [Agha 86]. Although we claim that processes scheduled using the reactive strategy will behave like Actors, we still have problems in guaranteeing weak fairness with fair infinite computations, specifically with the infinite production of messages. The reason for this anomaly is that Actor languages use syntactic rules to guarantee that the reaction to any single received message will take finite time, and that infinite message production can only be the product of the action of an infinite number of Actors. Our processes can be written in any sequential programming language; hence, a compile-time check is not applicable, since such infinite behavior can be a legal construct in a language at hand. Consider the following example:

```
while (TRUE) {
    if ( (m=xmalloc(n)) != NULL )
        xsend (m, node, pid);
}
```

The above computation is fair according to the definition from Section 3.1.2, and it is also a syntactically correct part of C program; however, this process does not satisfy the reactive property, because it will never terminate or execute an `xrecv`. Once this process is put in the *running* state, it will remain in that state indefinitely; therefore, scheduling on the node on which it resides will not be weakly fair.

We will describe a modification of the scheduling strategy that is sufficient to guarantee weak fairness in scheduling on a multicomputer node; the only requirement is that all processes on that node be fair.

The scheduling strategy is identical to that described in Section 3.2, except for the following state transition: When the process issues the `xmalloc` system call to request a message buffer, the process changes its state from *running* to *blocked* and a reply message of size equal to that of the requested message buffer is put at the tail of the receive queue (sent to the requester).

The only case in which the scheduling strategy defined in Section 3.1 cannot guarantee weak fairness even though all the processes involved are fair is the case of a process that will never either terminate or execute an `xrecv`, but that is still fair because it sends an infinite number of messages. To send any number of messages, a process needs at least that many message buffers. Since the only way to obtain a message buffer is to execute an `xrecv`, `xrecvrpc`, or `xmalloc`, the process will eventually exit the *running* state; this is sufficient to guarantee weak fairness.

3.4 Unfair Processes

Thus far, we have considered scheduling only in an idealized environment in which all processes on a certain node are fair, and our scheduling strategy has relied heavily on that fact.

It is quite possible, and happens frequently, that an unfair process will be spawned on a multicomputer node and scheduled to run. A programming error that causes the process to be wrapped in an endless loop without any communication action inside the loop body is an example of a situation that presents difficulties for the scheduler. As defined so far, the reactive scheduler does not have a way to handle this kind of error, and so an external mechanism is required to cope with this problem.

An important issue here is how to detect an error and then decide to preempt the *running* process; this work does not contain a clear answer to that question. The weak semantics of the weak fairness prevents us from placing any finite bound on the time that a process can be *running*. In real implementations, however, this problem is more of an artificial issue. We can use a process-selectable bound on the time that it will be *running*, and use a timer to enforce it. If the bound is exceeded, we can either assume an error and generate a restartable image of a process, or simply put the process into the *blocked* state, and insert a reply message for it in the receive queue (in front of any other reply message already inside the receive queue) that will be the token to run again.

3.5 Interrupt Messages, or, RK as a Message Processor

So far, we have assumed that all processes on a multicomputer node are equally (un)important. Because of this, the scheduler was able to employ a simplistic scheme, invoking the processes merely according to the message order in the receive queue.

However, the efficiency of implementation of an operating system requires making certain 'system' processes more important in some sense than user processes. For instance, a set of processes implementing a distributed file system would require higher execution priority than user processes, so that file system performance will not depend on the weak fairness of the reactive scheduling.

To deal with this issue, we now introduce the notion of *interrupt messages*, which have their own queue within a multicomputer node, and have execution priority over the normal messages. If we think of RK as being a software implementation of a message processor, then the effect of interrupt messages is analogous to that of hardware interrupts in an instruction processor.

To guarantee mutual exclusion, and for the same reasons that we usually disable interrupts within the interrupt handling routines in the instruction processor, processes that respond to normal messages should not be sent interrupt messages (and *vice versa*) (Figure 3.3).

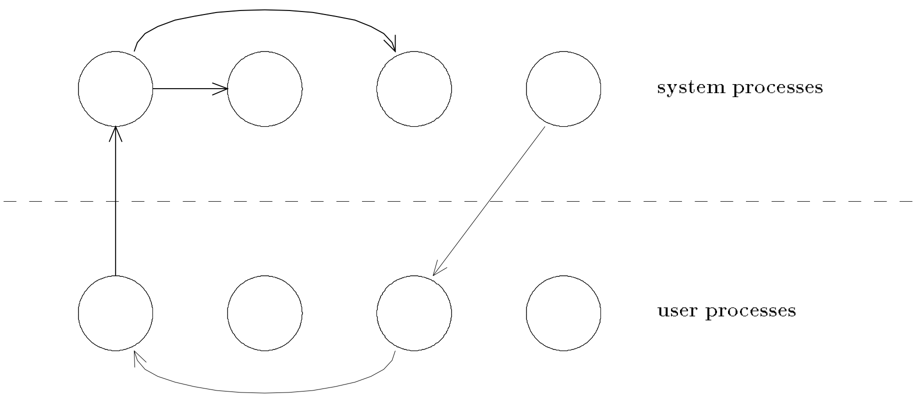

Figure 3.3: System processes receive only interrupt messages, and user processes only normal messages.

The only change to the scheduler is that it is given priority level information at the time that it is invoked, so as to decide which receive queue to operate on.

The interrupt-message concept can be generalized readily to a system with n levels of message priorities, corresponding to n interrupt levels. Again, any process must receive messages of only one priority level. This requirement enables us to make this scheme equivalent to the one using different-priority processes, rather than messages. The main reason for our preference for assigning priority levels to messages, rather than to processes, is that we do not want the message interface part (Section 4.3) of the multicomputer node to have any notion about the process structure.

3.6 Why Reactive Scheduling?

It might appear that we have gone to a lot of effort to develop a specific (reactive) scheduling strategy, only to discover that we will still have to put in an escape, in the form of an RPC, as well as something very similar to conventional, time-driven scheduling. Our main objective, however, was to design a scheduling strategy that would not penalize every computation with a constant penalty. We use the scheduling that carries the minimum overhead for the kinds of computations that occur most frequently in multicomputers; our design calls for rarely used features to pay the larger price.

This particular scheduling strategy has been implemented in the RK operating system, and it is described in Chapters 4 and 5.

Chapter 4

The Reactive Kernel

The Reactive Kernel is the product of an experiment in operating system design. Our approach was analogous to that used in the design of a RISC processor; RK employs simple and fast solutions for frequently used features. It is a portable operating system, suitable for running on both first- and second-generation multicomputers. Its layered structure, with the well-defined interfaces between the layers, provides for debugging and easy modifications and extensions.

The overall structure of the RK is illustrated in Figure 4.1. It consists of two major parts: the *Inner Kernel* (IK), and a set of *handlers* $\{H_0, H_1, \ldots, H_{n-1}\}$.

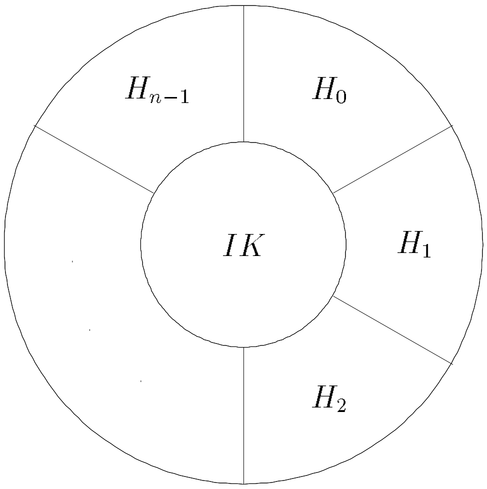

Figure 4.1: The RK structure

4.1 The Inner Kernel

The IK operates on only the following four types of objects: storage space, the send queue and the receive queue, messages, and handlers. There is no notion of a user; the user interface is provided by a dedicated handler, and no other part of the system has any information about that particular user interface.

The services provided by the IK can be grouped into two classes:

1. The dispatch mechanism, used by the IK to deliver the incoming messages to the handlers; and

2. The services performed on request, invoked by a handler executing the system call (sending a message, allocating storage).

4.1.1 The Dispatch Loop

From the standpoint of the IK, the handler is a function with an associated arbitrary data structure. The control flow in the system is very simple: The IK fetches the message from the receive queue and delivers it to the handler specified by the *tag* information in the message header. If there are no messages in the receive queue, handlers are selected in a round-robin fashion and given a NULL message. The inner loop of the IK is:

```
main ()
{
  MESSAGE *m;
  struct handler_desc {
    FUNC_PTR  f;
    DATA_PTR  d;
  } *h;

    while (1) {
        m = get_message (&h);
        (*(h->f)) (h,m);
    }
}
```

Once the control has been given to a handler, it executes until it terminates by executing a `return` statement. Thus, all operations performed by that handler can be considered to be a single atomic action. This strategy requires that every handler satisfy the *reactive property:* It has the obligation to terminate in a finite time and give control back to the dispatch loop.

4.1.2 System calls

Since the handlers are generally compiled separately from the IK, and linked as independent objects, a handler executes a system call to obtain system services from the IK. These services include sending messages, allocating storage, reporting errors, and obtaining information on the current multicomputer configuration.

As part of the system software, handlers are not accessible by the user; they run in the same privileged mode of operation used by the IK. Handlers operate only on variables from the stack and from the dynamically allocated space (the same as used by the IK); there is no data segment associated with a handler. Hence, handlers do not have to go through a context switch to access the IK routines; the system calls can be implemented with a IK-resident table used for indirect function calls.

If the IK is always loaded on the same absolute address, which is typically the case, the IK symbol table can be used at the handler link time, enabling handlers to access IK services with zero overhead.

4.2 The Handlers

4.2.1 Handler Environment

Except for the special case of a NULL message being delivered to a handler, the behavior of a handler is analogous to that of an *Actor* [Agha 86]. A handler is invoked when the message at the head of the receive queue is tagged for it; as a *reaction* to the message, it may *send* messages, create *new* handlers, and change its persistent internal state (*become* a new handler). By changing its state (including its entry point), a handler specifies its replacement behavior; *ie,* it can replace itself with an identical handler (default action), self-destruct, or become a new handler.

A handler is similar to the kernel processes in other operating systems, but, unlike the usual kernel process, no information is preserved for a handler by the IK between the two invocations; any information must be saved explicitly by the handler in its associated data structure, which resides in the dynamically allocated space. A similar approach of using light-weight kernel processes can be found in [Kale & Shu 88].

A handler may not be *killed* by any action other then self-destruction. Since the IK has no knowledge of the data structure associated with a handler, the handler itself is expected to do the cleaning-up task. If this property were not satisfied, the IK would have to do the garbage collection.

At boot time, the system consists of the IK and a single *spawn* handler, as illustrated in Figure 4.2. The *spawn* handler is used to spawn other handlers.

The set of handlers to be used in a particular node of a multicomputer will be determined at the time the node is allocated; this scheme provides the system with additional flexibility. In the typical space-sharing mode of operation [Seitz *et al.* 88a], we can have different handler configurations on different subsets of the nodes of a single multicomputer.

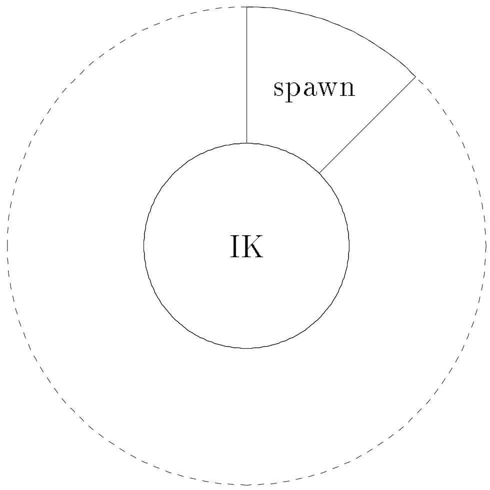

Figure 4.2: The RK structure at boot time

To support a particular programming language, or an application system, a handler and a library are all that are necessary. A handler supporting programming in C has been implemented and is described in Chapter 5.

4.2.2 An Example of Programming with Handlers

Figure 4.3 illustrates the process of spawning a new handler.

A handler (*rqst*) that is requesting that a new handler be spawned sends a message tagged for the *spawn* handler at the destination node. Having checked that the required resources are available, *spawn* creates a new *read* handler, and replies to *rqst*. The code for *read* handler is linked with the IK, just as the code for *spawn* is, but, unlike *spawn*, no instances of *read* exist in the IK-resident handler table at the boot time.

If the request has been granted, the *rqst* will send the code of the handler that is being spawned (or have it sent). After the code has been received, possibly in multiple messages, *read* sends the acknowledging message to *rqst*, and becomes *new* handler by changing the function-pointer field of the appropriate `handler_desc` data structure.

With the algorithm described above, the use of multiple instances of *read* handlers will permit the simultaneous spawning of more than one handler on the same node.

4.3 The Queue Management

A multicomputer node exchanges information with the outside world only via messages. Typically, the graph representing the connections between the multi-

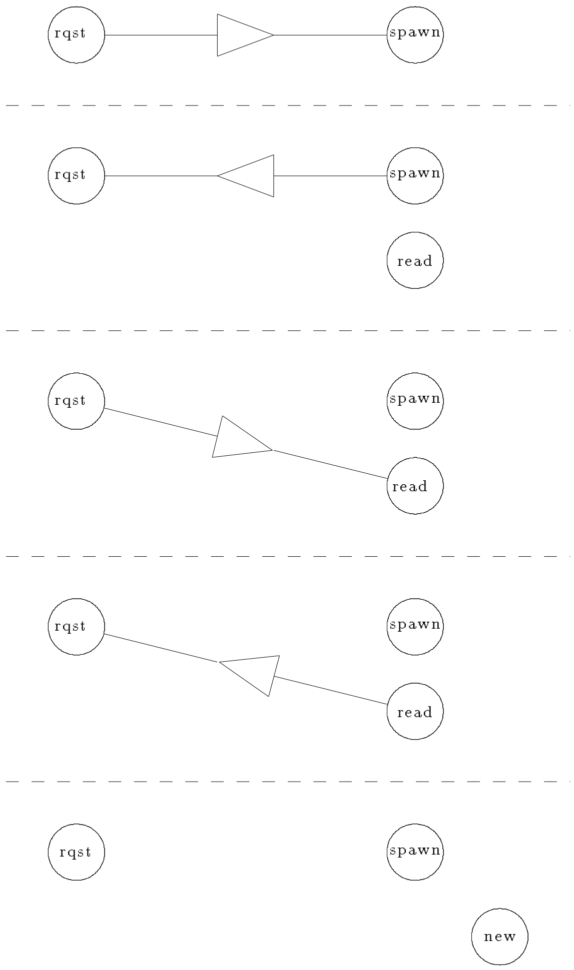

Figure 4.3: The spawning of a new handler

computer nodes is not fully connected, and message routing must be performed to support a programming model in which each process can communicate with every other process. Message routing is not part of the RK (Figure 4.4). In first-generation multicomputers, routing is performed by a set of interrupt routines; in second-generation multicomputers, it is done with hardware [Flaig 87]. In both cases, the routing subsystem interacts with the rest of the node via dedicated hardware or software that we will refer to as the *message interface* (MI).

The RK communicates with the message interface via the send and the receive queues. The send queue in general has multiple producers (handlers) and a single consumer (MI), but since all the operations performed by a handler — from invocation to termination — can be considered to be a single atomic action, the send queue management reduces to a single-producer, single-consumer case. The receive queue is analogous, where multiple producers are multiple communication channels, and the consumer is the IK dispatch loop. The task of maintaining the

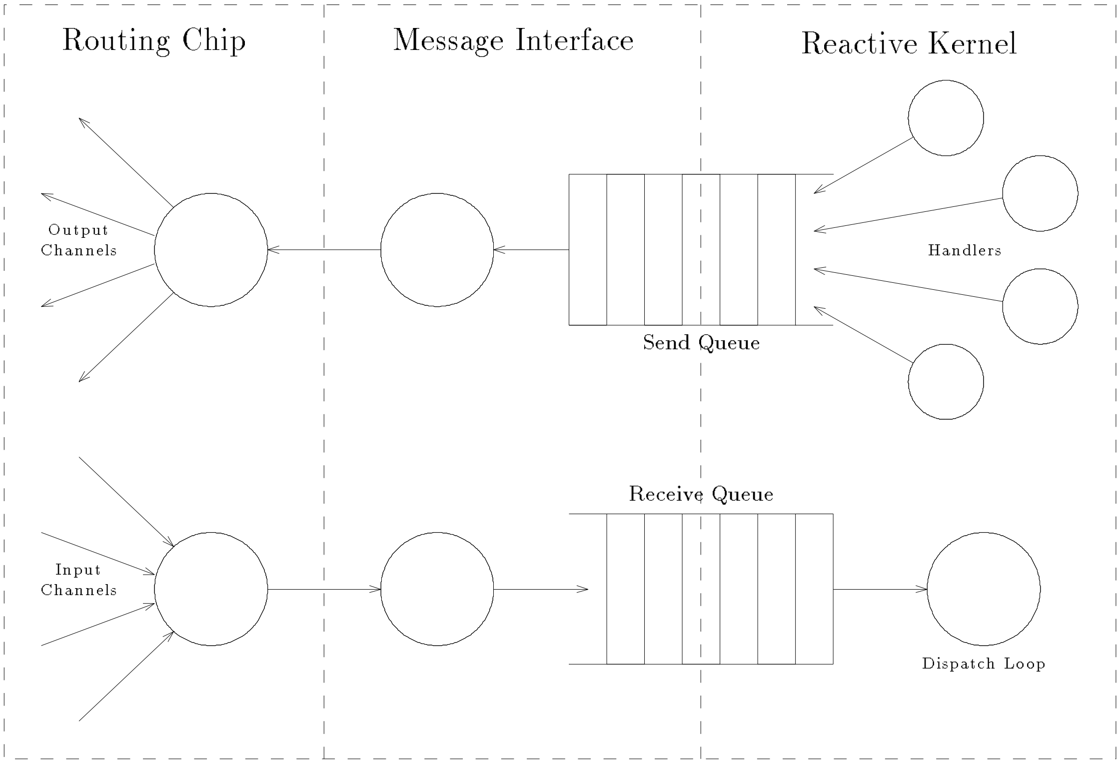

Figure 4.4: The structure of a multicomputer node

mutual exclusion of producers has to be performed by the routing subsystem or the MI.

Because the queue is shared by the RK and the MI, which operate concurrently, mutual exclusion has to be enforced. The single-producer, single-consumer queue can be efficiently implemented by using the data structure illustrated in Figure 4.5.

The queue consists of zero or more list elements with their end-flags equal to 0, and one last element with its end-flag equal to 1. The producer owns *put*, the pointer to the last element of the queue, and the consumer owns *get*, the pointer to the first element.

To put a message in the queue, the producer puts it into the last element, extends the queue with a new element with its end-flag equal to 1, and then resets the end-flag from the element that is no longer the last one.

The above algorithm can be proved correct; the only requirement is that the read and write operations on a single variable be atomic actions. The nice feature of the algorithm is that neither the *put* nor the *get* pointer is a shared variable; this reduces the interference between the producer and the consumer.

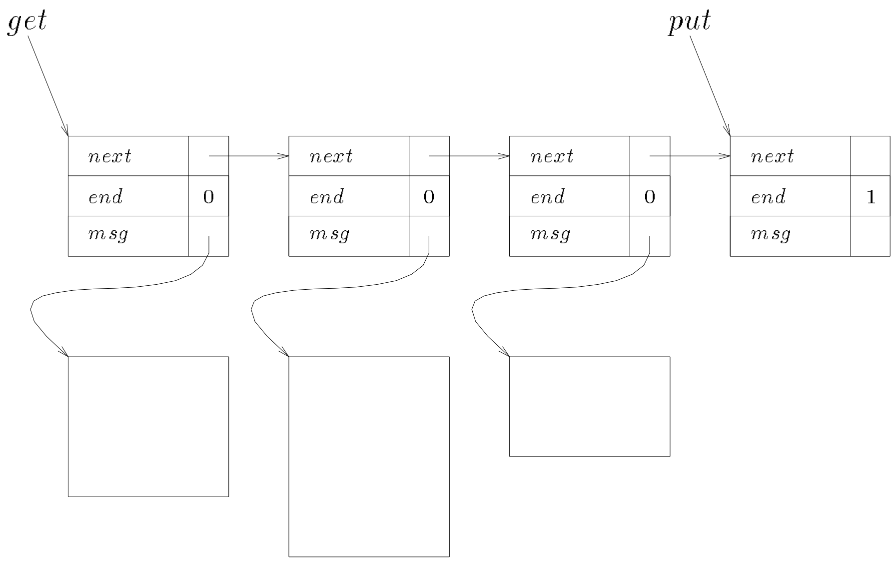

Figure 4.5: The structure of the queues

4.4 The Storage Allocation

In the RK environment, storage allocation is in the critical path that limits the communication performance of the multicomputer, and a significant part of the effort in implementing the RK is in storage-allocation strategy. Chapter 6 describes our implementation and gives an analysis of its performance.

Chapter 5

The User Interface

The support of programming at the user level, either in a particular programming language or within an application system, requires a user interface; this consists of a handler and a library. This chapter describes an implementation of the user interface for the C programming language.

The user interface performs the following tasks:

1. Dispatching incoming messages to user processes,

2. Servicing user system calls, and

3. Creating and terminating processes.

We have implemented the reactive process scheduling, including the additional RPC mechanism, as described in Section 3.2. The scheduling task itself is done implicitly by the first two tasks above. A process starts *running* when a message is dispatched to it, and it changes its state to *waiting* or *blocked* by specifying its choice points; these are special system calls.

5.1 Dispatch Mechanism

Because the control flow between the user-interface handler and the user processes it supports is analogous to the control flow between the Inner Kernel (IK) and handlers (Section 4.1.1), we refer to this dedicated handler as the *Reactive Handler* (RH).

An example of the data structure used by the RH is illustrated in Figure 5.1. It consists of a process table, a set of handler descriptors, and a set of process descriptors.

A system consisting of the RH and a set of handlers managed by the RH (no user processes) can be viewed as another level of recursive implementation of the model. A message, along with the right to use the processor, propagates down the handler tree until it reaches its final destination.

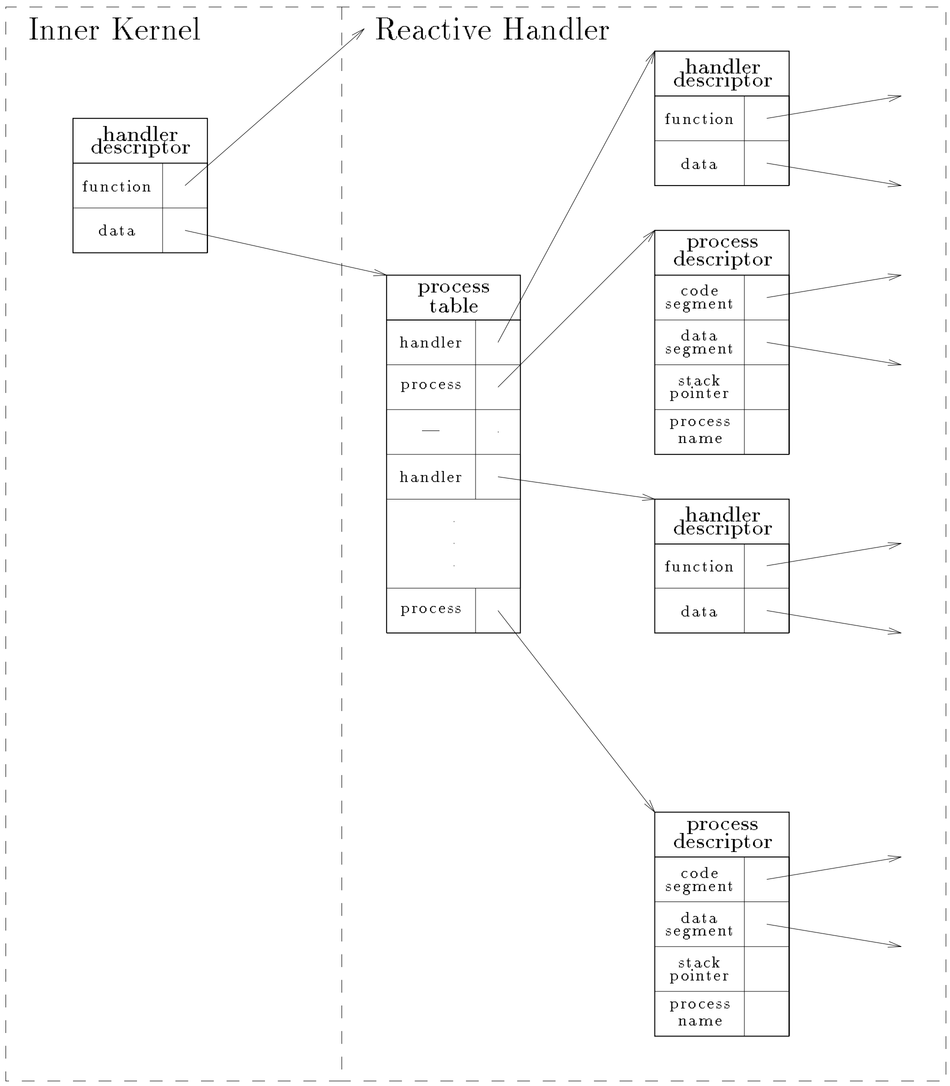

Figure 5.1: Structure of the Reactive Handler

The handlers managed by the RH are used to create and terminate processes, obtain information on the current state of a particular user process, and handle other operating-system services. Examples of handler usage can be found in Sections 5.2.1 and 5.3.

The reactive scheduling of user processes fits in this model; however, since both stack and static data have to be preserved for the user between two invocations, a user process cannot be called as a function; a context switch is required:

```
reactive_handler (h, m)
struct handler_desc *h;
MESSAGE *m;
{
```

```
        if (message_destination_is_a_handler(m))
            call_the_destination_handler (h, m);
        else
            run_the_destination_process  (h, m);

        return;
    }
```

Process descriptors contain all information necessary for performing a context switch in the particular hardware environment; their structure is implementation dependent.

5.2 System Calls

A user process executes system calls to obtain system services from the kernel. To keep the operating system design modular, system calls are serviced by the handler in the user interface, and the handler can in turn invoke IK services, if necessary.

At the user process link time, however, it is not known where the code (or data) of any particular handler will reside, since the handlers are themselves loaded at multicomputer-allocation time. A possible solution would be to have an IK-resident table of entries that would be filled up by the user-interface handler upon loading, and to have the user processes use that table to access the handler services. The same mechanism could be used to direct error handling to the RH, since it is typically the handler's job to handle all exceptions caused by the user processes that it manages.

This solution is not satisfactory when we have multiple user-interface handlers loaded at the same time, each supporting its own group of user processes. The table described above would have to be reinitialized on a per-message basis before a context switch could be executed to allow a user process to run. An acceptable solution introduces an additional level of translation: There is a single entry (or a few of them) in the IK that is modified by the user-interface handlers on a per-message basis, and each handler contains a table of entry points for the system services that it provides.

User processes execute two classes of system calls:

1. System calls that (as a side effect) specify choice points, and

2. System calls that do not affect the scheduling.

5.2.1 Choice-Point System Calls

The `exit`, `xrecv`, `xrecvb`, and `xrecvrpc` (and possibly `xmalloc`, as explained in Section 3.3) system calls are members of the first class. Upon any of these calls, control is given back to the RH, with an exactly inverse action to the context switch performed at the message-dispatch time. Depending on the system call executed, a process will be killed (`exit`), put into the *waiting* state (`xrecv`), or put into the *blocked* state (`xrecvrpc`). The `xrecvb` function can be implemented as a library function using the `xrecv`, or a process that executes `xrecvb` can be put into a *waiting* state and marked as not willing to accept the NULL message.

Except for `xrecvrpc`, the implementation of all system calls in this class can be done with the message-dispatching mechanism described in Section 5.1. During the RPC, however, non-reply messages have to be saved on a queue associated with the *blocked* process. We use the following implementation for the RPC: An entry in the process table corresponding to the *blocked* process is changed to point to a handler descriptor of a special *wait* handler. In this way, each message destined for the *blocked* process will instead be delivered to its associated *wait* handler. The *wait* handler is responsible for filtering out the non-reply messages and managing the queue of them.

5.2.2 Regular System Calls

All remaining system calls, as defined in [Seitz *et al.* 88a], are in the second class. Within this class, system calls that involve the CE/RK services that are not resident on the same node with the requesting process (`spawn`, `ckill`, `print`, and `execute`) are implemented using the RPC mechanism.

Implementation of the calls for services that can be resolved within the node is straight-forward (`xsend`, `xsendrpc`, `xfree`, `xlength`, `mynode`, `mypid`, `nnodes`, `cubedim`, `clock`, and `led`).

5.3 Process Creation and Termination

The process-spawning mechanism is analogous to the handler-spawning mechanism described in Section 4.2.2. If there are multiple instances of the same process on a single node, the code segment is shared. To avoid unnecessary access to the file system, the initialized part of the data segment is kept with the code, and copied to the data segment of each new process instance.

A special *ckill* handler is a member of the set of handlers managed by the RH, and its job is to terminate user processes. It has a higher priority then do user processes, and accepts interrupt messages (Section 3.5); thus, it can be used to terminate user processes without waiting for them to exit the *running* state.

5.4 Time-Driven Scheduling

The mechanism for falling back on time-driven scheduling for long-running processes described in Section 3.4 has been implemented. The handling of the timer interrupt is redirected to the RH routine as shown in Section 5.2. A user process is preempted if it does not leave the *running* state during its time-slice, and a reply message is put into the receive queue for it (in front of any other reply message already there) as a token to run again.

Chapter 6

Storage Allocation

This chapter includes a fairly detailed analysis of a solution for a specific problem in operating system design. A reader not interested in the details of the operating system implementation may skip Section 6.4.

6.1 Storage Requirements

The choice of storage allocation strategy for RK depends on a number of parameters: the frequency of the requests for allocation and deallocation, the distribution of the sizes of the requested blocks, the time between allocation and deallocation of a particular block, the protection requirements.

In RK, there are three major storage classes:

1. The storage used for the user process code and data;

2. The storage required by the RK for queues, message descriptors, process descriptors; and

3. The storage where the messages reside.

The first class is characterized by infrequent allocation and deallocation requests, which occur only during process creation and termination. Block sizes in this storage class are typically the largest in the system, and the configuration remains unchanged for the longest time periods. These characteristics suggest that the allocation strategy should be as efficient as possible in terms of storage utilization, preferably without fragmentation, even if that requires a time-inefficient algorithm.

The most important requirement for the second class is speed. Most of the system services provided by the RK will use the storage from this class. Typically, the requested block size is the smallest in the system, and the blocks' lifetimes are the shortest. The allocation scheme for constant-size blocks, with constant allocation and deallocation times, is the most appropriate for this class.

The third class is midway between the first two, in respect to the block size as well as the frequency of allocation requests. The message size typically ranges from a few bytes to several kilobytes; the time that messages spend in the system is determined by the user, and ranges from a few tens of CPU instructions to the lifetime of a process. This storage class requires an efficient algorithm for allocating blocks of different sizes.

6.2 Protection

The protection requirements for the first two storage classes do not differ from those in any other multiple-process operating system and are not the subject of our interest in this work.

Because the storage pool in which the messages reside has to be accessible by each process on a node, its protection requirements are more restricted. The two most frequently used hardware-enforced protection mechanisms in today's machines are based on assigning access rights to *segments* or *pages*. In the first approach, the process is assigned a set of segments, generally of arbitrary sizes, whose number is limited by the number of segment registers in the processor. In the second approach, the process is assigned an arbitrarily large set of fixed-size pages. The segment approach is not applicable to the problem since a process can at any time have an arbitrary number of references to messages. The paging mechanism can be employed by assigning a number of pages to each message.

A number of algorithms for storage allocation exist in the literature; the particular one used for each of the classes will depend on the hardware organization of the computer (and *vice versa*). No particular memory allocation scheme was considered to be sufficiently general and portable for inclusion in the RK specification; rather, an interface that consists of a number of functions for storage allocation and deallocation is defined, and this interface is used as a part of the design specifications for porting of the RK.

6.3 The Back-Reference Problem

Various algorithms for the storage allocation of different-size blocks and the analysis of their efficiency can be found in [Knuth 68]. All of them have in common the feature that for each used block of memory, there is an associated data structure, called a *descriptor*, that contains the relevant information about the corresponding block. This scheme creates a *back-reference* problem; *ie,* given the pointer to a particular memory block, how to find the appropriate descriptor.

An obvious solution is to keep the descriptor pointer or the whole descriptor within the memory block itself; however, this solution is an unsatisfactory, since the misuse of pointers by user processes could cause an operating system error.

What is needed is a *dictionary*, *ie,* a set representation with the *insert*, *delete*, and *member* operations. The set elements are descriptors, and the *keys* are pointers to the memory blocks.

Let N_{max} be the number of different keys, $0 \leq key < N_{max}$, and N_{msg} be the number of used memory blocks in the memory. One possible solution for implementing a dictionary is to maintain a linear list of all used memory blocks. The memory required is minimal, $O(N_{msg})$, the time required for the insert operation is $O(1)$, but the time required for member and delete operations is $O(N_{msg})$. The other extreme is to have an array indexed by the key. In this case, all operations on the set can be done in $O(1)$ time, but the memory required is $O(N_{max})$.

6.4 A Solution for the Back-Reference Problem

The following analysis evaluates the performance of the compromise solution; the algorithm used is based on [Morrison 68]. The idea of the algorithm is as follows: To access an element of the set, we perform *digital searching* [Knuth 73] along the N-ary tree for k steps, whereby with each step we reduce the number of possible elements by a factor of N. After k steps, we are left with at most $n = \frac{N_{max}}{N^k}$ possible outcomes, and we resolve the remaining ambiguity by the sequential search. The linear list solution is the special case of this scheme for $k = 0$, and the array solution is the special case for $N = N_{max}, k = 1$.

Figure 6.1 illustrates the data structure used by the algorithm for one set of parameters. The leaves of the N-ary tree are linear lists of the elements of the set, containing at most n elements. The structure shown corresponds to the worst case, when all N_{max} possible elements are in the set. We will refer to the graph that represents the set containing the maximum number of elements as the *complete tree.*

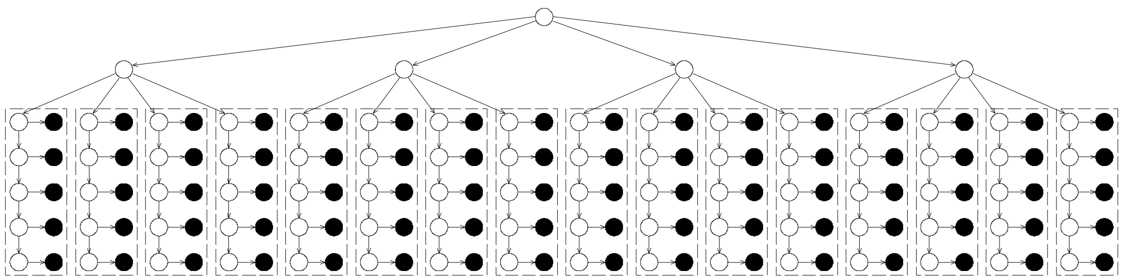

Figure 6.1: Set representation for $N = 4, k = 2, N_{max} = N_{msg} = 80, n = 5$

A node of the tree exists if, and only if, it has at least two children, and all other nodes are 'skipped'. An example subset structure is shown in Figure 6.2.

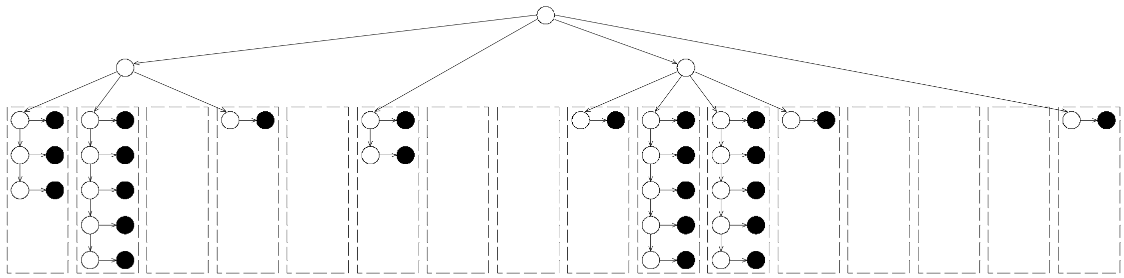

Figure 6.2: An example subset representation

The set of nodes in the path to a leaf is dependent on all elements currently in the set, but not on the previously performed operations on the set.

Performance of the algorithm clearly depends on the number of elements in the set; this corresponds to the number of messages in our system. To evaluate that dependence, we will first do an analysis of a hypothetical system in which all messages are of the same size; later, we will show how the distribution of message sizes affects the obtained results.

6.4.1 Terminology

The general case is represented in Figure 6.3.

The level information is embedded in the node and does not change. It does not necessarily correspond to the length of path from the root to that node, except in a complete tree.

To simplify the notation, we will assume that the N-ary tree does not change dynamically. However, all the nodes that are not present (*ie,* 'skipped') in the actual implementation will be considered *dead*; *ie,* they have zero memory requirements, and take zero time to be operated on. All other nodes are *live*.

We will say that a fixed node or edge *covers* a leaf if, and only if, it is in the path from the root to that leaf. Therefore, a node at the level $i, 0 \leq i < k$ covers N^{k-i} leaves, and each of its outgoing edges covers N^{k-i-1} leaves.

Let N_{max} be the number of available *slots* in which messages can reside, where each leaf of the tree has $n = \frac{N_{max}}{N^k}$ slots. All slots and all messages are of unit size. Let N_{msg} be the number of messages present in the slots. When a new message arrives, it is randomly placed into any of the free slots, with equal probability: $\frac{1}{N_{max}-N_{msg}}$. A message is released after an arbitrary length of time that does not depend on the slot in which it resides.

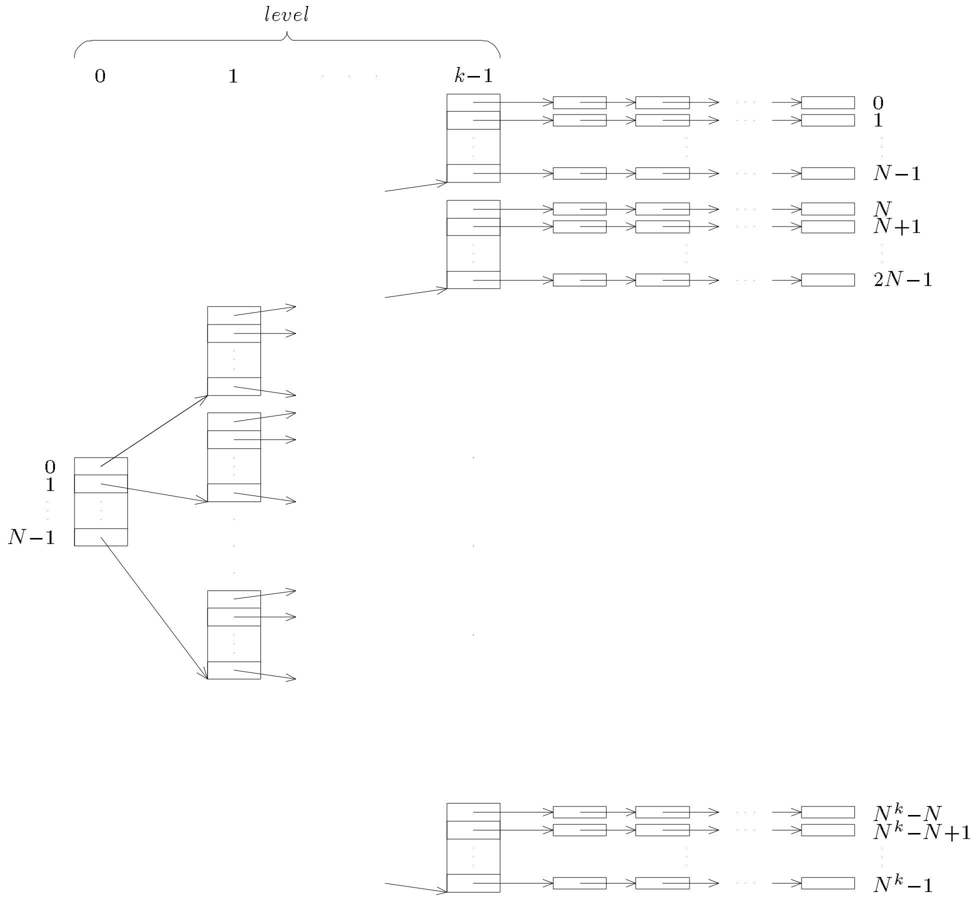

Figure 6.3: General case set representation

Then, if there are N_{msg} messages in the system, each of the $C_{max} = \binom{N_{max}}{N_{msg}}$ possible ways to choose which N_{msg} out of N_{max} slots is used is equally probable. We will refer to each one of these choices as a *configuration*.

Throughout the analysis, we will assume that

$$\binom{n}{k} = \begin{cases} \frac{n!}{k!(n-k)!} & , n \geq k \\ 0 & , n < k \end{cases} . \tag{6.1}$$

6.4.2 Space Complexity

Let $\mathcal{N}$ be a fixed node at level $i, 0 \leq i < k$. Let $P_{dead}(i, N_{msg})$ be the probability that $\mathcal{N}$ is dead, given that the number of messages in the system is N_{msg}. Let $C_{dead}(i, N_{msg})$ be the number of configurations for which $\mathcal{N}$ is dead. We are averaging across all configurations, so

$$P_{dead}(i, N_{msg}) = \frac{C_{dead}(i, N_{msg})}{C_{max}}. \tag{6.2}$$

The node $\mathcal{N}$ is dead if, and only if, there is at least one message for at most one of its outgoing edges in the slots that that edge covers:

$$C_{dead}(i, N_{msg}) = N\binom{N_{max} - (N-1)\frac{N_{max}}{N^{i+1}}}{N_{msg}} - (N-1)\binom{N_{max} - \frac{N_{max}}{N^i}}{N_{msg}}. \quad (6.3)$$

In the first term, we count the configurations with all slots covered by at least $N-1$ outcoming edges of $\mathcal{N}$ being empty. We should also take into account configurations for which there are no messages in any of the slots covered by $\mathcal{N}$, but this has already been included N times in the first term, so the second term makes a necessary adjustment.

Let $K_{avg}(N_{msg})$ be the average number of nodes in the tree. Averaged across all possible configurations,

$$K_{avg}(N_{msg}) = \sum_{i=0}^{k-1} N^i(1 - P_{dead}(i, N_{msg})). \quad (6.4)$$

In the worst case,

$$K_{max} = \sum_{i=0}^{k-1} N^i = \frac{N^k - 1}{N - 1}. \quad (6.5)$$

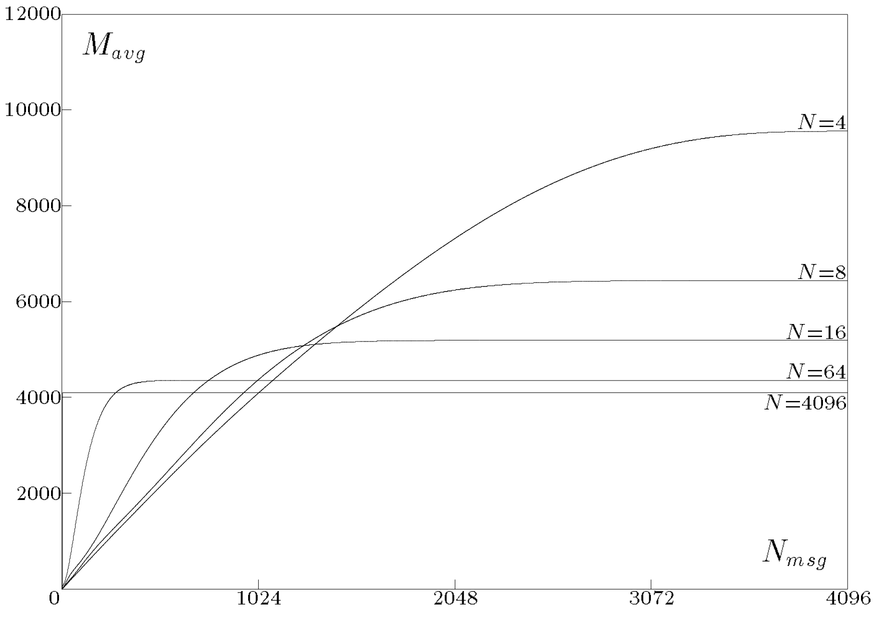

Figure 6.4: Memory requirements for $N_{max} = 4096, k = \log_N N_{max}, k_1 = 3, k_2 = 0$

The memory required by a single node is $N + k_1$, where k_1 is the constant overhead, and the memory used by an entry in the linked list is k_2. The average memory requirements of the algorithm are

$$M_{avg}(N_{msg}) = (N + k_1) \cdot K_{avg}(N_{msg}) + k_2 \cdot N_{msg}, \quad (6.6)$$

with the maximum

$$M_{max} = (N + k_1)\frac{N^k - 1}{N - 1} + k_2 N_{max}. \tag{6.7}$$

6.4.3 Time Complexity

Let $\mathcal{M}$ be any one fixed message in the system with a total of N_{msg} messages. The time required to access $\mathcal{M}$ is proportional to the number of live nodes we have to go through in order to reach it. Let $P_{skipped}(i, N_{msg})$ be the probability that a node $\mathcal{N}$ at level i that covers $\mathcal{M}$ is dead, and $C_{skipped}(i, N_{msg})$ be the number of configurations for which this is the case. These configurations occur when for all outgoing edges of $\mathcal{N}$, except the one covering $\mathcal{M}$, there are no messages in the slots they cover.

$$C_{skipped}(i, N_{msg}) = \binom{N_{max} - 1 - (N-1)\frac{N_{max}}{N^{i+1}}}{N_{msg} - 1} \tag{6.8}$$

$$P_{skipped}(i, N_{msg}) = \frac{C_{skipped}(i, N_{msg})}{\binom{N_{max}-1}{N_{msg}-1}} \tag{6.9}$$

The average length of the path from the root to the leaf containing $\mathcal{M}$ is

$$L_{avg}(N_{msg}) = \sum_{i=0}^{k-1} (1 - P_{skipped}(i, N_{msg})). \tag{6.10}$$

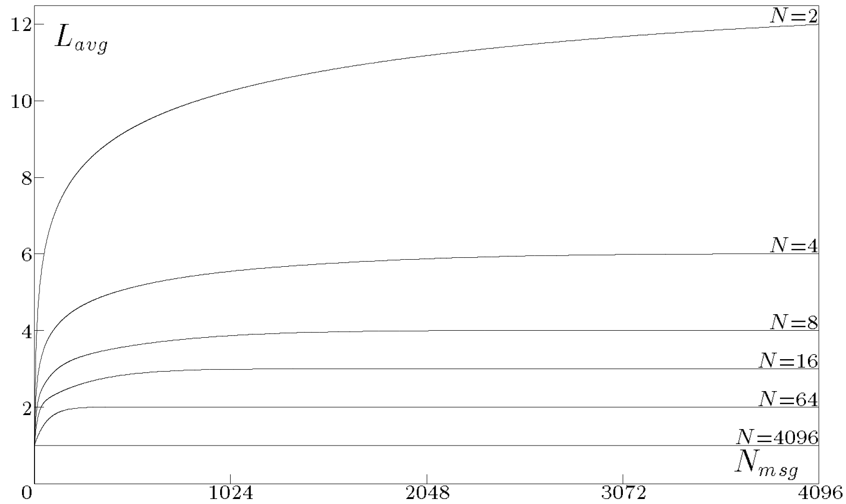

Figure 6.5: Average tree depth for $N_{max} = 4096, k = \log_N N_{max}$

Let $P_{length}(i, N_{msg})$ be the probability that there is exactly $i, (1 \leq i \leq n)$ messages in the list containing $\mathcal{M}$.

$$P_{length}(i, N_{msg}) = \frac{\binom{N_{max}-n}{N_{msg}-i}}{\binom{N_{max}-1}{N_{msg}-1}} \tag{6.11}$$

The probability that we will have to go through exactly $d, (1 \leq d \leq n)$ levels in order to reach $\mathcal{M}$ is

$$P_{depth}(d, N_{msg}) = \sum_{i=d}^{n} \frac{1}{i} P_{length}(i, N_{msg}), \tag{6.12}$$

and in the average case we will have to go through $D_{avg}(N_{msg})$ levels to reach a message:

$$D_{avg}(N_{msg}) = \sum_{d=1}^{n} d\, P_{depth}(d, N_{msg}). \tag{6.13}$$

On average, the total time is proportional to

$$T_{avg}(N_{msg}) = L_{avg}(N_{msg}) + D_{avg}(N_{msg}), \tag{6.14}$$

and the maximum time is

$$T_{max} = k + n = k + \frac{N_{max}}{N^k}. \tag{6.15}$$

Tree-Reconfiguration Cost

Finally, let us look at how often we have to change the configuration of the tree; *ie,* how often a node changes the state from dead to live or *vice versa.*

Let us assume that there are N_{msg} messages in the system, and that we want to discard message $\mathcal{M}$. Let $\mathcal{S} = \{\mathcal{N}_i | 0 \leq i < k\}$ be the set of nodes, where $\mathcal{N}_i$ is a node at the level i in the path from the root to the leaf in which $\mathcal{M}$ resides. Let j be the largest integer such that $\mathcal{N}_j$ is live. This means that one of the pointers in the $\mathcal{N}_j$ is pointing to $\mathcal{M}$, and $\mathcal{N}_i, j+1 \leq i < k$ are skipped since they are dead. When we deallocate $\mathcal{M}$, it may happen that we also need to change the state of $\mathcal{N}_j$ to dead. This will happen if $\mathcal{M}$ is the only message in the slots covered by one of the outgoing edges of $\mathcal{N}_j$, and if exactly one of its remaining $N-1$ outgoing edges has at least one message in the slots it covers. Let $\mathcal{E}$ be the edge from $\mathcal{N}_{j-1}$ to $\mathcal{N}_j$. If the deallocation resulted in 'killing' $\mathcal{N}_j$, then there were at least two messages covered by $\mathcal{E}$ prior to deallocation; therefore, after the deallocation, there will be at least one message covered by it. This means that the state change from live to dead does not propagate towards the root; *ie,* at most one of the nodes in the tree can change its state during the deallocation. The probability that the node at the level i will change its state from live to dead when deallocating a message from the system with N_{msg} is

$$P_{change}(i, N_{msg}) = \frac{(N-1)\binom{N_{max}-(N-1)\frac{N_{max}}{N^{i+1}}}{N_{msg}-1} - (N-1)\binom{N_{max}-\frac{N_{max}}{N^i}}{N_{msg}-1}}{\binom{N_{max}-1}{N_{msg}-1}}, \qquad (6.16)$$

and the probability that we will have to reconfigure the tree upon deallocating a message is

$$P_{dealloc}(N_{msg}) = \sum_{i=0}^{k-1} P_{change}(i, N_{msg}). \qquad (6.17)$$

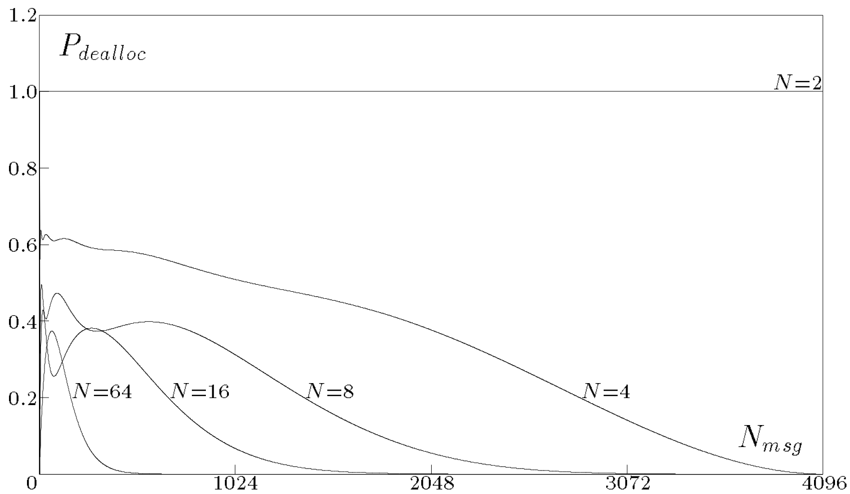

Figure 6.6: Tree-reconfiguration probability for $N_{max} = 4096, k = \log_N N_{max}$

Since the configuration of the tree does not depend on the order in which allocations and deallocations are performed, the probability that we will have to change the configuration upon allocating a message in the system containing N_{msg} messages is

$$P_{alloc}(N_{msg}) = P_{dealloc}(N_{msg}+1). \qquad (6.18)$$

6.4.4 The Effect of the Distribution of Message Sizes

To get more realistic results, we will look at the performance of the above-described algorithm on message sizes of a given distribution. Let $\mathcal{M}$ be a *multiset* of positive integers, $|\mathcal{M}| = N_{msg}$, each integer representing the size of one message in the system. There are a total of N_{max} unit-size slots in the system, and a message of size s can be placed in any s consecutive free slots. We will calculate $P_{start}(i, \mathcal{M})$, which is the probability that a slot i is the bottom of any message in a system containing N_{msg} messages with the given message-size distribution described with the multiset $\mathcal{M}$.

Let $\mathcal{S}_{\mathcal{M}} = \{s_i | 0 \leq i < z\}$ be a set of positive integers obtained by removing all duplicate elements from $\mathcal{M}$. Let $n_i, 0 \leq i < z$ represent the number of occurrences of s_i in $\mathcal{M}$. Let $sum(\mathcal{A})$ be the function defined on multisets such that $sum(\mathcal{A}) = \sum_{j \in \mathcal{A}} j$.

To find $C(\mathcal{A}, n)$, which is the number of ways to place the messages represented by a multiset $\mathcal{A}$ into the n consecutive free slots, we will define multiset $\mathcal{Z}_{\mathcal{A}}$, which consists of $n - sum(\mathcal{A})$ elements all equal to zero (one for each unused slot) and multiset $\mathcal{A}_0 = \mathcal{A} \cup \mathcal{Z}_{\mathcal{A}}$. Then $C(\mathcal{A}, n)$ is equal to the number of permutations of $\mathcal{A}_0$:

$$C(\mathcal{A}, n) = \begin{cases} \frac{|\mathcal{A}_0|!}{|\mathcal{Z}_{\mathcal{A}}|! \prod_{i=0}^{z-1} n_i!} & , n \geq sum(\mathcal{A}) \\ 0 & , n < sum(\mathcal{A}) \end{cases} = \begin{cases} \frac{\prod_{i=1}^{|\mathcal{A}|} (|\mathcal{Z}_{\mathcal{A}}| + i)}{\prod_{i=0}^{z-1} n_i!} & , n \geq sum(\mathcal{A}) \\ 0 & , n < sum(\mathcal{A}) \end{cases} . \tag{6.19}$$

Let $\mathcal{P}(\mathcal{A})$ be the set of all possible, distinct, ordered pairs $(\mathcal{P}_0(\mathcal{A}), \mathcal{P}_1(\mathcal{A}))$, such that $\mathcal{P}_0(\mathcal{A})$ and $\mathcal{P}_1(\mathcal{A})$ are obtained by partitioning the multiset $\mathcal{A}$ into two multisubsets.

The probability that there is a message of size s starting at slot k in the system with N_{max} slots and messages represented by the multiset $\mathcal{M}$ is

$$P_0(s, k, \mathcal{M}) = \frac{\sum_{\mathcal{P}(\mathcal{M} - \{s\})} (C(\mathcal{P}_0(\mathcal{M} - \{s\}), k) \cdot C(\mathcal{P}_1(\mathcal{M} - \{s\}), N_{max} - s - k))}{C(\mathcal{M}, N_{max})}, \tag{6.20}$$

and the probability that there is a message of any size starting at slot k is

$$P_{start}(k, \mathcal{M}) = \sum_{s \in \mathcal{S}_{\mathcal{M}}} P_0(s, k, \mathcal{M}). \tag{6.21}$$

Probability $P_{start}(k, \mathcal{M})$ has been calculated for uniform, linear, quadratic, and exponential message-size distributions, for a wide variety of values for N_{max} and N_{msg} parameters. There is no noticeable effect due to message-size distribution. A typical case is illustrated in Figure 6.7.

With the assumption that the size of a single message does not exceed a few percent of the storage pool wherein the messages reside (which is typically the case), $P_{start}(k, \mathcal{M})$ differs from N_{msg}/N_{max} less then 1%, except for the edge effects. This makes our results for the space and time complexity a good approximation in the case of non-unit-size messages.

6.4.5 Logical Addresses → Block Descriptors

Since most second-generation multicomputers will support some form of virtual memory, the original specification of the back-reference problem has to be slightly modified: Given the logical address of a memory block, we have to find the block

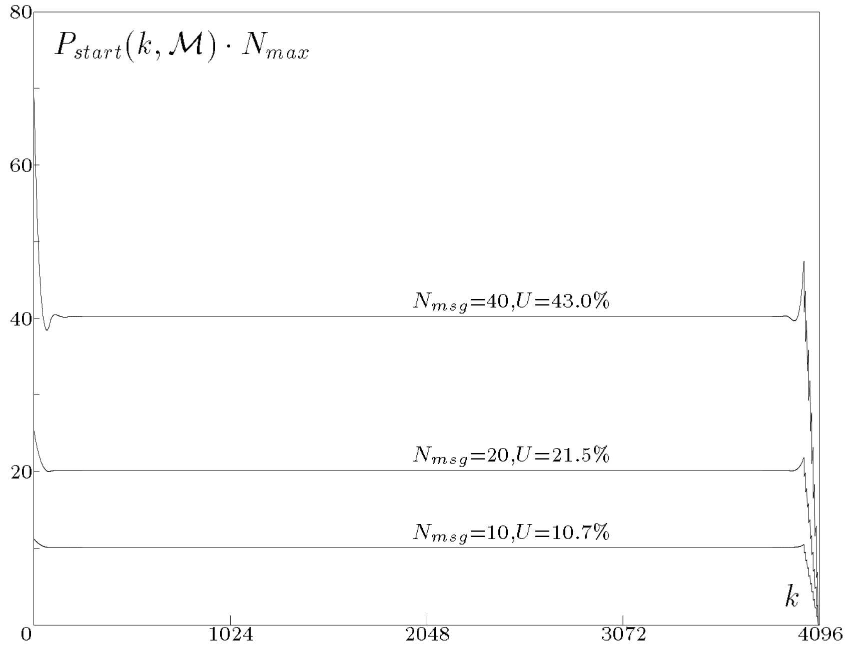

Figure 6.7: Probability that a message starts at slot k. Message-size distribution is uniform; maximum message size is equal to 80; $N_{max} = 4096$. The U parameter represents the used portion of the message pool.

descriptor of the corresponding physical block. One possible solution is to do the logical-to-physical address translation first, and then use the physical address as a key for the dictionary we have described.

An alternative approach is to increase the size of the set that we are working with from N_{max} to $N_{max} \cdot N_{context}$, where $N_{context}$ is the number of different logical-to-physical maps that the hardware supports. In this case, the key for the dictionary would be the logical address extended with the field indicating the map used.

Which approach should be used has to be determined by comparing the additional searching cost with the savings due to eliminating the logical-to-physical address translation.

Chapter 7

Results and Future Work

7.1 Summary

The work presented in this report is an experiment in operating system design. It was motivated primarily by the desire to better utilize the performance of the second-generation multicomputers.

When the design of the RK got started, we created a wish list. Here is what we wanted the RK to be, together with what we think the RK became:

- *simple* — We identified the key features of a multicomputer operating system, and tried for a minimalistic solution. The approach used in the design of the RK is much like that of a RISC processor, employing simple and fast solutions for frequently used features. This approach led us to the simplistic scheduling strategy and streamlined message handling. The resulting kernel is very small and simple, and still provides the full set of functions needed in a multiple-task node operating system.

- *portable* — The implementation-dependent parts of RK (memory allocation, context switching) are well isolated from the rest of the design. The port of the code developed on the Cosmic Cube to the Ametek Series 2010 was achieved without any serious difficulties, while retaining about 90% of the original code. The port to the Intel iPSC/II is in progress.

- *modular* — The careful layering of the RK structure, with well-defined interfaces between the layers, provides for easy modifications and extensions. RK can be tested and tuned by incrementally adding more-complex features without interfering with the already tested ones.

- *fast* — Although we have reduced the number of context-switches to, at most, one per message, made the cost of a single context-switch equal to that of the ordinary system function call, and decreased the number of CPU instructions in message-handling by about a factor of three, the software component is still the dominant part of message latency. Part of the reason for this comes from the necessity of satisfying protection requirements, and could be eliminated if, for instance, the code for user processes were a result of a compilation procedure that could guarantee no misuse of message pointers. However, any additional significant decrease in the software component of the message latency would require architectural support.

7.2 Where Next?

7.2.1 Handlers as the Compilation Target for High-Level Languages

Initially, the idea of splitting the RK into two fairly independent parts — the Inner Kernel and the set of handlers — came about from a realization that separating the concerns would make the implementation easier, and make future modifications straight forward. However, W.C. Athas' use of the handler environment as a compilation target for the Cantor programming notation, and the work of L.V. Kale and W. Shu on the *Chare-Kernel* [Kale & Shu 88], have convinced us that elaborating on the handler environment will be beneficial.

7.2.2 Multiple-User Support

So far, we have used multicomputers in a space-sharing environment in which each user allocates a number of multicomputer nodes. Although there may be more then one process on a node, all belong to that same user. Commercial second-generation multicomputers already support hardware protection for multiple-user programming, and shortly will provide distributed file systems, and virtual memory on the nodes. RK can easily provide for multiple users without any modifications; each user will have a personal handler to manage the set of processes belonging to that user.

Still lacking is a way to manage resources in a multicomputer. In an ideal environment, all software would be written for multicomputers, including editors, compilers, *etc*, and organized as a collection of many relatively small cooperating processes, that would be mapped automatically onto the available nodes by the underlying kernel mechanism. The smaller the processes, the easier the problem of load balancing becomes, and we can do reasonably well with random process placement, provided that the communication latency does not depend heavily on the distance between communicating nodes. This is already true for second-generation multicomputers.

Bibliography

[Agha 86] G.A. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986.

[Aho *et al.* 74] A.V. Aho, J.E. Hopcroft, J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[Athas 87] W.C. Athas, *Fine Grain Concurrent Computation*, Caltech Computer Science Technical Report [5242:TR:87], 1987.

[Athas & Seitz 88] W.C. Athas, C.L. Seitz, *Multicomputers: Message-Passing Concurrent Computers*, IEEE Computer, pp. 9–24, August 1988.

[Dally 87] W.J. Dally, *A VLSI Architecture for Concurrent Data Structures*, Kluwer Academic Publishers, 1987.

[Dally & Seitz 87] W.J. Dally, C.L. Seitz, *Deadlock-Free Message Routing in Multiprocessor Interconnection Networks*, IEEE Transactions on Computers, pp. 547–553, May 1985.

[Flaig 87] C.M. Flaig, *VLSI Mesh Routing Systems*, Caltech Computer Science Technical Report [5241:TR:87], 1987.

[Kale & Shu 88] L.V. Kale, W. Shu, *The Chare-Kernel Language for Parallel Programming: A Perspective*, University of Illinois at Urbana-Champaign Technical Report [UIUCDCS-R-88-1451], 1988.

[Knuth 68] D.E. Knuth, *The Art of Computer Programming: Fundamental Algorithms*, Addison-Wesley, 1968.

[Knuth 73] D.E. Knuth, *The Art of Computer Programming: Sorting and Searching*, Addison-Wesley, 1973.

[Morrison 68] D.R. Morrison, *PATRICIA — Practical Algorithm To Retrieve Information Coded in Alphanumeric*, Journal of the ACM, pp. 514–534, 1968.

[Seitz 84] C.L. Seitz, *Concurrent VLSI Architectures*, IEEE Transactions on Computers, pp. 1247–1265, Dec. 1984.

[Seitz 85] C.L. Seitz, *The Cosmic Cube*, Communications of the ACM, pp. 22–33, Jan. 1985.

[Seitz *et al.* 88a] C.L. Seitz, J. Seizovic, W.-K. Su, *The C Programmer's Abbreviated Guide to Multicomputer Programming*, Caltech Computer Science Technical Report [CS-TR-88-1], 1988.

[Seitz *et al.* 88b] C.L. Seitz, W.C. Athas, C.M. Flaig, A.J. Martin, J. Seizovic, C.G. Steele, W.-K. Su, *The Architecture and Programming of the Ametek Series 2010 Multicomputer*, Proceedings of the 1988 Hypercube Conference, 1988.

[Su & Seitz 86] W.-K. Su, C.L. Seitz, *Object-Oriented Event-Driven Simulation* in *Submicron Systems Architecture Semiannual Technical Report*, pp. 15–16, Caltech Computer Science Technical Report [5235:TR:86], 1986.

[Su *et al.* 87] W.-K. Su, R. Faucette, C.L. Seitz, *C Programmer's Guide to the Cosmic Cube* Caltech Computer Science Technical Report [5252:TR:87], 1987.