RENDEZVOUS PRIMITIVES FOR
OPERATING SYSTEM DESIGN

Charles Dean Pevsner

TR 81-471
January 1982

Department of Computer Science
Cornell University
Ithaca, New York
RENEDEVOUS PRIMITIVES FOR OPERATING SYSTEM DESIGN

A Thesis
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment for the Degree of
Master of Science

by
Charles Dean Pevsner
January, 1982
Abstract

A well chosen, comprehensive set of kernel primitives is essential for the construction of higher levels of a computer's operating system. In particular, although many solutions to the classical operating system problems of interprocess synchronization and communication have been proposed in the past, difficulties with destination naming and language construct symmetry have continued to mar most proposed solutions.

This thesis proposes a set of primitives, the Rendezvous primitives, that form the kernel of an operating system. It is argued that the Rendezvous primitives constitute a reasonable and practical set of operations on which to base an operating system. The Rendezvous primitive itself is presented as a solution to the interprocess synchronization and communication problems. A rendezvous occurs when two processes synchronize and subsequently exchange messages. Rendezvous is symmetric, in that processes that wish to communicate both use the same primitive; and processes invoke Rendezvous with class designations, not procedure names. Rendezvous is presented within the context of the other primitives, which support the restriction of access to Rendezvous, process creation and destruction, process scheduling, and stack resource management. A comparison of Rendezvous and existing primitives and language constructs to solve interprocess synchronization and communication is made.

The full syntax and semantics of the Rendezvous primitives are presented, along with a discussion of their design rationale. The Crossbar
Switch, a generalized virtual device interconnection and reconfiguration scheme, is presented as an example of the use to which the Rendezvous primitives might be put. A high level program description of the Crossbar Switch is presented as well.

The thesis concludes with a description of the implementation of the Rendezvous kernel on a DEC PDP 11/60. The kernel code is given in an appendix.
Biographical Sketch

Charles Dean Pevsner was born on June 20, 1958 in Los Angeles, California. He graduated from Fremd High School, in Palatine, Illinois, in 1976. He received the Bachelor of Arts degree in English from Cornell University in 1980, graduating with distinction. He is a member of Phi Beta Kappa and Tau Beta Pi.
Acknowledgments

My advisors, Professor Alan Demers and Professor Fred Schneider, deserve the bulk of the credit for this work. Their perspicacity and ingenuity have time and again rescued this project from intellectual deadlock. In particular, Professor Schneider deserves credit for being the first to suggest Rendezvous as a structuring idea. Professor Demers is responsible for more brilliant proposals than I can even begin to list. Most of all, I want to thank him for his encouragement and support, from our initial Terminal Switch discussions, to those 10pm-8am debugging nights.

I owe special thanks to Dr. Joseph Bates for going out of his way to get me involved in this project in the first place. He was a patient and sage counselor, and his friendship and moral support have brought me through the fingernail-biting times.

Many other individuals have contributed in important ways to this work, although they may not have known it. Robert Harper and Professor Sam Toueg took time to read early specifications, listen to presentations, and argue about the primitives and Crossbar Switch. Francis Corrado's wit and exuberance were clear tonics to my mind and nerves.

I cannot adequately express my thanks for the ceaseless support of my parents throughout this time.
Table of Contents

1. Introduction .................................................. 1

1.1 The Rendezvous primitives ............................... 1

1.2 Organization of the thesis ............................... 1

1.3 Description of the primitives ........................... 2

  1.3.1 Synchronization and communication ................ 2

  1.3.2 Process creation and destruction .................. 4

  1.3.3 Interrupt handling ................................. 4

  1.3.4 Scheduling ......................................... 4

  1.3.5 Memory conservation ............................... 4

1.4 What’s different about Rendezvous ..................... 5

  1.4.1 Semaphores ....................................... 5

  1.4.2 Monitors ......................................... 6

  1.4.3 Communicating Sequential Processes ............... 6

  1.4.4 Distributed Processes ............................ 8

  1.4.5 Ada Entry and Accept ............................. 8

1.5 Features of the Rendezvous primitives ................ 9

2. The primitives and their rationale ..................... 11

  2.1 Overview ............................................. 11

  2.2 The meta-language ................................... 11

  2.3 Rendezvous classes and types ........................ 12

  2.4 The Rendezvous primitive ............................ 13

    2.4.1 Syntax ......................................... 13

    2.4.2 Semantics .................................... 14
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4.3</td>
<td>The symmetry of Rendezvous</td>
<td>16</td>
</tr>
<tr>
<td>2.4.4</td>
<td>Why rv_classes are typed</td>
<td>17</td>
</tr>
<tr>
<td>2.4.5</td>
<td>The disjunctive rendezvous</td>
<td>18</td>
</tr>
<tr>
<td>2.4.6</td>
<td>Conditions: Inside or outside the primitive?</td>
<td>18</td>
</tr>
<tr>
<td>2.4.7</td>
<td>Binding time</td>
<td>20</td>
</tr>
<tr>
<td>2.4.8</td>
<td>Returned values</td>
<td>21</td>
</tr>
<tr>
<td>2.5</td>
<td>Acquiring and releasing rendezvous classes</td>
<td>21</td>
</tr>
<tr>
<td>2.5.1</td>
<td>Syntax</td>
<td>21</td>
</tr>
<tr>
<td>2.5.2</td>
<td>Semantics</td>
<td>22</td>
</tr>
<tr>
<td>2.5.3</td>
<td>Dynamic Create and Release</td>
<td>23</td>
</tr>
<tr>
<td>2.5.4</td>
<td>Keeping track of rv_classes</td>
<td>23</td>
</tr>
<tr>
<td>2.6</td>
<td>Spawning a child process</td>
<td>24</td>
</tr>
<tr>
<td>2.6.1</td>
<td>Syntax</td>
<td>24</td>
</tr>
<tr>
<td>2.6.2</td>
<td>Semantics</td>
<td>25</td>
</tr>
<tr>
<td>2.6.3</td>
<td>How to spawn</td>
<td>25</td>
</tr>
<tr>
<td>2.7</td>
<td>Treating an interrupt as a rendezvous</td>
<td>26</td>
</tr>
<tr>
<td>2.7.1</td>
<td>Syntax</td>
<td>26</td>
</tr>
<tr>
<td>2.7.2</td>
<td>Semantics</td>
<td>27</td>
</tr>
<tr>
<td>2.7.3</td>
<td>Interrupt handlers</td>
<td>27</td>
</tr>
<tr>
<td>2.8</td>
<td>Timed rendezvous</td>
<td>28</td>
</tr>
<tr>
<td>2.8.1</td>
<td>Non-blocking rendezvous</td>
<td>29</td>
</tr>
<tr>
<td>2.8.2</td>
<td>The clock manager</td>
<td>29</td>
</tr>
<tr>
<td>2.9</td>
<td>Setting a process stack size</td>
<td>31</td>
</tr>
<tr>
<td>2.9.1</td>
<td>Syntax</td>
<td>32</td>
</tr>
<tr>
<td>2.9.2</td>
<td>Semantics</td>
<td>32</td>
</tr>
<tr>
<td>2.9.3</td>
<td>Frugal memory organization</td>
<td>32</td>
</tr>
<tr>
<td>2.10</td>
<td>Setting a process priority</td>
<td>32</td>
</tr>
</tbody>
</table>
2.10.1 Syntax ........................................... 32
2.10.2 Semantics ........................................ 33
2.10.3 Scheduling ........................................ 33

3. Sample application: The Crossbar Switch ................. 36
  3.1 Overview ........................................... 36
  3.2 Drawbacks to standard configurations ................. 36
  3.3 The Crossbar Switch approach ....................... 38
  3.4 The CS/device interface ........................... 40
  3.5 The CS model ....................................... 40
    3.5.1 Connections and connection classes ............ 41
    3.5.2 Division of labor ............................. 43
  3.6 Operation of the CS ................................ 44
    3.6.1 The connection protocol ....................... 44
    3.6.2 Port process/Connection process interaction ... 46
    3.6.3 Disconnect protocol ............................ 47
  3.7 CS code ........................................... 47
  3.8 Alternative designs ................................ 47

4. Implementation of the Rendezvous Kernel .................. 58
  4.1 Overview ........................................... 58
  4.2 Interface to C ..................................... 58
    4.2.1 Create ........................................ 59
    4.2.2 Release ....................................... 60
    4.2.3 Rendezvous .................................... 61
    4.2.4 Fork ........................................... 63
    4.2.5 Exit .......................................... 63
    4.2.6 Assoc .......................................... 63
4.2.7 Stksize ........................................ 64
4.2.8 Setpri ......................................... 64

4.3 Memory management and process switching ............... 65
4.3.1 Physical memory organization ...................... 65
4.3.2 Virtual memory organization ...................... 66
4.3.3 System calls and process switching ............... 67
4.3.4 Interrupts, traps, and more on process switching ... 71

4.4 Data structures .................................. 72
4.4.1 Runlist ....................................... 73
4.4.2 Process Entries ................................ 74
4.4.3 Rendezvous Classes ............................ 75
4.4.4 Rendezvous Table Entries ....................... 76
4.4.5 Association Table ............................. 77

4.5 How the primitives work ........................... 77
4.5.1 create() ................................... 77
4.5.2 release() ................................... 78
4.5.3 stksize() ................................... 79
4.5.4 setpri() ................................... 79
4.5.5 fork() ....................................... 80
4.5.6 exit() ....................................... 80
4.5.7 assoc() ..................................... 80
4.5.8 rendezvous() .................................. 81
4.5.9 interrupt() .................................. 84

4.6 Memory management routines .......................... 85

4.7 Suggestions for improvements .......................... 85

Appendix A. Specification for the Rendezvous Primitives .... 88
List of Illustrations

Figure 2.1 Bounded Buffer Manager, using UncondRV ...................... 19
2.2 Bounded Buffer Manager, using Rendezvous ...................... 20
2.3 Establishing a Private Communication Line ...................... 24
2.4 The Spawn Operation ...................................... 26
2.5 An Interrupt Handler Prototype ............................. 28
2.6 Timer Usage .............................................. 29
2.7a Clock Manager: Initial Exchange .............................. 30
2.7b Clock Manager: Child Startup ................................ 31
2.7c Clock Manager: Software Tick ................................. 31
2.7d Clock Manager: User Wakeup ................................ 31
2.8 The Clock Manager ........................................... 34
3.1 Conventional Backend Machine/tty Configuration ............... 37
3.2 Crossbar Switch Connections .................................. 39
3.3 Logical Constructs in the 11/60 ................................ 41
3.4 A Connection Across the CS ................................... 42
3.5 Connection Manager .......................................... 43
3.6 Connection in Progress ........................................ 44
3.7a Connection Manager .......................................... 49
3.7b Connection Process ......................................... 52
3.7c Port Manager Process ........................................ 54
3.7d Port Process ................................................ 56
4.1 Physical Memory Organization .................................. 65
4.2 User Virtual Address Map ..................................... 66
4.3 Kernel Virtual Address Map ........................................... 67
4.4 After a Call from Process A ......................................... 69
4.5 Before Returning to Process B ....................................... 71
4.6 List Header Structure .................................................. 73
4.7 Rendezvous Table Snapshot ........................................... 83
Chapter 1

Introduction

1.1. The Rendezvous primitives

A well chosen set of primitive operations can make the construction of higher levels of a computer's operating system a pleasant and rewarding task. If, on the other hand, a system designer makes a poor choice of kernel primitives, then the remainder of the project becomes a clumsy attempt to ameliorate the effects of that early decision. This thesis proposes a set of primitives, the Rendezvous primitives, that form the kernel of an operating system. The Rendezvous primitives comprise a clean approach to the classical operating system headaches of interprocess synchronization and communication, process creation and destruction, and scheduling, and provide a smooth interface with external devices. We believe that the Rendezvous primitives allow higher levels of an operating system to be written in a clear, concise, and even elegant manner.

1.2. Organization of the thesis

The rest of this chapter contains a brief description of the Rendezvous primitives, and describes how the central primitive for dealing with
synchronization and communication differs from existing primitives or language constructs. An attempt is made to summarize what we feel are the chief favorable features of the Rendezvous primitives.

Chapter 2 gives a full syntactic and semantic description of the primitives, along with their design rationale and usage examples. Chapter 3 describes a sample operating system application, the Crossbar Switch, and shows how the Rendezvous primitives can be used to implement the Crossbar Switch. Chapter 4 describes our implementation of the Rendezvous kernel in the C programming language on a DEC PDP 11/60, and suggests some improvements for future implementations. The section of Chapter 4 that describes the user/kernel C interface, along with the semantic description of the primitives in Chapter 2, comprise a "user's manual" for the Rendezvous kernel.

Appendix A collects in one place the raw syntactic and semantic description of the primitives scattered throughout Chapter 2. Appendix B contains the Rendezvous kernel code.

1.3. **Description of the primitives**

1.3.1. **Synchronization and communication**

The three primitives that deal with interprocess synchronization and communication are:

--- **Create**: Gives the caller capabilities for the Rendezvous primitive. In general, a capability is defined as the right of access to an object. Create gives the caller the right to use the Rendezvous primitive.
-- **Release**: Allows the caller to relinquish capabilities.

-- **Rendezvous**: Effects a synchronization and message exchange (a rendezvous) between two processes.

When a process wants to talk to or wait for another process, it requests a rendezvous on one or more rendezvous classes. Rendezvous classes are created in pairs with Create, and each member of the pair is said to match the other member. A rendezvous occurs when two processes invoke Rendezvous using matching rendezvous classes.

A Rendezvous request is disjunctive, in that a process that specifies several rendezvous classes is requesting a rendezvous on any one of several classes. However, only one rendezvous will take place per call, no matter how many rendezvous classes are specified in the call.

If the Rendezvous request is not immediately successful, then the calling process blocks until it is. That is, if two processes want to talk, the first to invoke Rendezvous will block. When the second process invokes Rendezvous on the matching rendezvous class, the first caller is awakened, and messages are exchanged between the two processes. A single process cannot deposit a message and continue on its way; Rendezvous is not a mailbox primitive.

A caller can, as we have said, specify as many rendezvous classes in a Rendezvous call as desired. The caller specifies a message along with each rendezvous class in the call. If a rendezvous takes place on a given class, the corresponding message is passed to the other process. In addition, the caller can include boolean expressions with each rendezvous class and message pair, dictating whether or not the class and message should be
included in the Rendezvous request.

1.3.2. Process creation and destruction

Two primitives govern the birth and death of new processes:

-- **Fork**: Clones an identical copy of the calling process. The new clone has different memory management registers, and a different return value, but otherwise is an exact copy.

-- **Exit**: Kills the calling process.

A newly created process has a copy of all of its parent's capabilities, which it can then use to rendezvous. Chapter 2 discusses a higher-level Spawn operation, built from the rendezvous primitives, that can be used to restrict what a cloned process inherits from its parent.

1.3.3. Interrupt handling

The **Assoc** primitive causes an interrupt from a hardware device to be treated as a Rendezvous call. This allows interrupts from devices to be treated in the same fashion as signals from processes.

1.3.4. Scheduling

The **Setpri** primitive allows processes to specify the software priority at which they should be run. Higher priority processes run before lower priority processes, and processes at the same level run in round-robin order.

1.3.5. Memory conservation

The **Size** primitive allows a process to specify the size of its
stack area. When a process is created with Fork, it inherits the same stack size as its parent. Stksize allows adjustment of that stack size.

1.4. What's different about Rendezvous

1.4.1. Semaphores

A good many other proposals for dealing with the problems of interprocess synchronization or communication or both have been proposed over the years, as these problems have remained particularly obdurate to solution. Semaphores [Dijkstra 68] allow processes to synchronize effectively and communicate with mutually excluded access to a shared data area. Unfortunately, semaphores have the drawback that one can forget to use them. This causes processes to block forever, critical sections to be accessed by more than one process, and like disasters. Forgetfulness is less likely with the rendezvous primitives, since invoking Rendezvous is the only way to synchronize or communicate.

In addition, the P and V operations on semaphores provide no direct way to test the value of a semaphore. Thus, there is no way to avoid blocking on a P operation if the value of the semaphore is 0. Although the Rendezvous primitives provide no direct way to examine a rendezvous class to see if a process is waiting on it, one of two approaches can be used to effectively perform the test. First, a Rendezvous request is disjunctive, and the order in which argument rendezvous classes are examined for matches is left-to-right. Thus, if a rendezvous occurs on the third class specified in a call, the caller knows that no process was waiting on either of the first two classes. The second approach is to build a "test rendezvous class" operation on top of the rendezvous kernel. This approach is similar
to the construction of a non-blocking Rendezvous operation, described in Chapter 2.

1.4.2. Monitors

We have designed Rendezvous to solve the interprocess communication and synchronization problems within the context of the entire kernel. This makes it difficult to directly compare Rendezvous with other primitives and constructs that address synchronization and communication. For instance, the monitor concept [Hoare 74] combines critical sections with synchronization signals. Our primitives implement a kernel in which every process running above the kernel has its own data area. The intent is that since no common data areas exist, the critical section problem does not arise.

It may be that a system requires global access to a resource or data area. This does not rule out the use of the rendezvous primitives. The critical stuff could managed by a single process, which services requests from other processes to use the resource or data area. A user process would request use of the resource by performing a Rendezvous with the resource manager, passing it a description of the required service as its part of the Rendezvous message exchange. Such a server process superficially resembles a monitor, in that critical sections of code are collected in one area, and only one process at a time (viz., the server) is allowed to access any critical section. However, no data are actually being directly shared.

1.4.3. Communicating Sequential Processes

The Communicating Sequential Processes (CSP) language construct [Hoare 78] provides for a one-way message transfer between two processes. Each
process must name the other, and specify whether it is the source or the
destination for the message transfer. Synchronization is effected by
blocking the first process to reach its send or receive statement. This
aspect of CSP resembles the operation of Rendezvous, as it does its re-
striction that processes may not communicate by accessing common data areas.

However, just as an interprocess signal can be thought of as a degener-
ate case of a message transfer, so can a one-way message transfer between
two processes be thought of as a degenerate case of a message exchange.
Rendezvous effects a message exchange, but can be used to pass one-way mes-
sages or simply signal. Rendezvous classes are typed, in that each class
allows messages of only one type to be passed. Use of a rendezvous class
of type void causes a rendezvous on that class to act as a signal instead
of passing a message.

In CSP, two processes that wish to communicate must do so by name.
Rendezvous, on the other hand, will effect a message exchange with any pro-
cess holding the matching capability. A particularly nice feature of Ren-
dezvous is that a capability may be passed as a Rendezvous message. This
allows users to hold a global rendezvous capability for a system server,
rendezvous with the server, and receive in exchange a capability for a
private communication line with the server. This kind of dynamic creation
of private communication lines is a powerful and attractive aspect of Ren-
dezvous, as will be illustrated in Chapter 2.

CSP uses Dijkstra's guarded commands to conditionally select a send or
receive statement on which to synchronize. Rendezvous conditionally
includes or discards argument list rendezvous classes in an analogous
manner. The structure of a CSP program is that of boolean conditions
guarding program blocks. In a program using Rendezvous, the conditions would appear in the Rendezvous argument list itself, and the program blocks would follow in a _case_ statement.

1.4.4. **Distributed Processes**

Brinch Hansen's concept of Distributed Processes [Brinch Hansen 78] allows processes to communicate by calling each other's parameterized internal procedures. A message is passed through receipt of an argument or a procedure return value. Rendezvous differs from the Distributed Processes concept in syntax and philosophy. Rendezvous effects a data exchange, not a one-way transfer. Furthermore, in keeping with the symmetry of a message exchange, Rendezvous is a symmetric language construct; both ends of message exchange use Rendezvous. With Distributed Processes, the sender and receiver use _send_ and _rec_ statements to effect a message transfer.

As with CSP, Distributed Process callers must name the internal procedures of other processes. Rendezvous does not require explicit naming, as communication requests are made through dynamically created capabilities.

1.4.5. **Ada Entry and Accept**

The Ada programming language strongly influenced the design of Rendezvous, although any similarities now seem remote. In particular, the term "rendezvous" comes from Ada. Ada includes entry calls and matching accept statements. Synchronization is achieved in same manner as with Rendezvous: the first process to execute an entry or accept blocks. Communication is effected as follows:
When an entry has been called and a corresponding accept statement is reached, the sequence of statements, if any, of the accept statement is executed by the called task (while the calling task remains suspended). This interaction is called a rendezvous. Thereafter, the calling task and the task owning the entry can continue their execution in parallel. [Ada 80]

Like Rendezvous, Ada allows a message exchange; unlike Rendezvous, however, the calling and the called process perform different actions. The Rendezvous primitive does not distinguish between caller and called processes. Ada has the calling process pass an argument, then remain suspended until the called process completes some task; after receiving a return value, the caller then proceeds on its way. With Rendezvous, the first process to request a rendezvous is suspended. However, once the rendezvous occurs, the message exchange takes place immediately.

1.5. Features of the Rendezvous primitives

We feel that the main advantages that the Rendezvous primitives have to offer include:

-- **Message exchange.** Rendezvous allows simultaneous message passing, often resulting in fewer exchanges than would be necessary with a one-way transfer primitive.

-- **Interrupt handling.** Devices are smoothly incorporated into the conceptual structure of a system. An interrupt from a device is treated as a call to the Rendezvous primitive. Interrupt handlers use the same primitives as other user processes.

-- **Low-level primitives.** The primitives do not unnecessarily restrict the user’s power; for instance, a Spawn primitive can be built from Fork.
-- Scoping of capabilities. Rendezvous classes are dynamically created and destroyed, helping to insure that capabilities are extant only as long as they are needed.

-- Typing of messages. The typing of Rendezvous messages allows pure signaling to take place, and supports restricts on capability propagation.

-- No explicit naming. A process need not know the name of the process to rendezvous with it. This allows the construction of library and server routines.

Bad features of the Rendezvous primitives will no doubt be brought to our attention in time.
Chapter 2

The Primitives and their Rationale

2.1. Overview

In this chapter we describe the full syntax and semantics of the rendezvous primitives, and discuss the rationale behind their design. The primitives were motivated chiefly by their possible applications, so we have not been hesitant to include short high-level program segments illustrating some of the finer points of the usage of the primitives. In this way we also intend to support our claim that the rendezvous primitives constitute a reasonable set of operations on which to base an operating system.

2.2. The meta-language

We use a small, straightforward metalanguage to describe the primitives. Consider this example:

\[ \text{Rendezvous}( \langle \text{cond}, \text{rv\_class}, \text{msg} \rangle^+ ) \]

Lower case letters indicate variable names. **Boldface** letters indicate keywords for the rendezvous primitives. The angle brackets, \(<\), indicate a group of one or more variables. The variable name `foo` by itself is equivalent to \(<\ foo\ >\), while \(<\ foo,\ foobar\ >\) means "the two variables `foo` and `foobar". If \(<\ foo,\ foobar\ >\) appears as the target of a assignment
statement, it does not mean that the r-value of the assignment is of the
type "pair of variables"; it means that the primitive on the right hand
side of the assignment returns two values.

The Rendezvous primitives can be treated as expression language con-
structs, so one need not target the result of a primitive call. In program
examples, we will sometimes omit values that we don't care about; for
instance, we might write < , b , > instead of < a , b , c > if only b is of
concern.

Finally, ( )+ means, "one or more repetitions of the entity between
the braces".

2.1. Rendezvous classes and types

A rendezvous class (or rv_class, for short) is a capability used to
exchange messages between processes. We define a capability as the right
of access to an object. Possession of a rv_class capability by a process
gives that process the right of access to the Rendezvous primitive, i.e.,
allows the process to request a rendezvous.

Note that capabilities and rv_classes are in reality distinct; rv_classes are "owned" by the rendezvous kernel, and individual processes
acquire only the capability to request a rendezvous on a rv_class. How-
ever, as far as an individual process is concerned, the capability in the
rv_class, since when it owns a capability to a rv_class, it can request a
rendezvous on that class. A process might own two different capabilities
that map to the same rv_class. In that case, the process perceives the
capabilities as two extensionally equal rv_classes. We shall usually be
viewing the primitives from the user process point of view, so unless the
distinction is important, we shall speak of capabilities and \textit{rv_classes} interchangeably.

\textit{Rv_classes} are created in pairs, and each member of the pair is said to \textit{match} the other member of the pair. For a message exchange to take place, two processes must request a rendezvous with matching \textit{rv_classes}. That is, one process must request a rendezvous with one member of the \textit{rv_class} pair, and another process must request a rendezvous with the matching member of the \textit{rv_class} pair.

Each \textit{rv_class} allows messages of one \textit{type} to be exchanged. There are currently three types:

\textbf{Data}. The message passed is a one-byte datum.

\textbf{Void}, meaning that no data are passed; the fact that a successful rendezvous took place is in the message.

\textit{Rv_class} is considered a type. Thus, \textit{rv_classes} may be passed in a message. The receiving process can use that \textit{rv_class} to make subsequent rendezvous requests. The sending process retains full use of the \textit{rv_class} as well. There is no notion of revocation of \textit{rv_class} capabilities; once process A has passed a \textit{rv_class} to process B, process A has no control over the use or propagation of that \textit{rv_class} by process B.

2.4. The \textit{Rendezvous primitive}

2.4.1. Syntax
Rendezvous( \{ < cond, rv_class, msg > \}^{+} )

Returns: \{ trip_num, return_msg, ok \}

2.4.2. Semantics

A rendezvous call is a request for a message exchange. A caller specifies a set of rendezvous classes, along with a message for each class. When the kernel receives two matching rv_classes from two separate calls (a rendezvous), then it swaps the corresponding messages between the two calling processes.

The kernel is passed one or more triples and returns three values. Each argument triple contains:

- **cond**, a boolean expression involving any variables available to the calling process.
- **rv_class**, identifying a rendezvous class.
- **msg**, a message to be passed to the other end of the rendezvous connection. The type of msg is determined by the rv_class.

The kernel associates a **triple number** with each triple. The triple number is simply the position of the triple in the argument list, counting from zero. That is, the leftmost triple is triple number 0, the next is triple number 1, and so forth.

The conditions in all the triples are evaluated at the time of the call in unspecified order. Those triples having conditions that evaluate to **false** are disregarded. When we refer to triples in the rest of this section, unless otherwise noted we are referring to only those triples hav-
ing conditions that evaluate to true.

The system kernel maintains the rendezvous table, a list of unmatched rendezvous requests. The kernel examines the triples in left to right order. For each triple, the kernel checks the rendezvous table to see if any other caller has requested a rendezvous with the matching rv_class of the triple. If no match is found for any triple, the triple number, rv_class, and msg from all triples are entered into the rendezvous table, and the calling process blocks.

Now consider the case in which a match is found for the \( n \)th triple. If the calling process is A, then triple \( n \) matches triple \( m \) in a previous call by process B. The triple number, the rv_class, and the message \( (msg_B) \) from triple \( m \) are in the rendezvous table. The kernel will:

1) If the type of \( msg_A \) is not void, return \( msg_A \) to process B (the return_msg value).

2) Wake up process B and return to B (the trip_num value).

3) If the type of \( msg_B \) is not void, return \( msg_B \) to process A (the return_msg value).

4) Return \( n \) to process A (the trip_num value).

5) Remove all of process B's rendezvous requests from the rendezvous table.

If the returned value ok is < 0, then the Rendezvous call was erroneous, and a rendezvous did not take place. Possible errors include: 1) For one or more triples the calling process does not own the specified rv_class.
2) All conditions have evaluated to false at the time of the call.

2.4.3. The symmetry of Rendezvous

The ADA programming language defines a rendezvous as the synchronization of two parallel tasks in order to effect an exchange of information. [Ichbiah et al. 79] After the synchronization and exchange take place, each process continues on independently. Rendezvous is essentially a symmetric event, in that each participant in the event follows the same algorithm. If two lovers are to rendezvous in the park, they both follow the same algorithm to effect the rendezvous:

1) Go to the park.
2) Is my lover here? If not, wait.
3) My lover is now here. Exchange sweet nothings.
4) Leave and go my separate way.

Despite the symmetric nature of a rendezvous, the ADA designers shied away from a symmetric approach to rendezvous; instead, ADA contains the asymmetric accept statement and entry call constructs. Although rendezvous does involve some asymmetry, in that typically two objects of opposite polarity intend to meet, we feel that the fact that both participants follow the identical algorithm demands a symmetric language construct. Hence, we have only one primitive, Rendezvous, to effect interprocess synchronization and communication. Synchronization is accomplished by having the first process to request a rendezvous block and wait for its partner; communication takes place when a rendezvous occurs, and messages are exchanged between the two processes.
There is no syntactic or algorithmic asymmetry in the Rendezvous primitive itself. Asymmetry arises in the concept of a rendezvous class, because the two members of a matching rv_class pair are opposites of sorts. We feel that there is no reason for the syntax to describe the polarity of a message transfer; indeed, there is no reason for a message exchange not to take place. Hence, Rendezvous effects a message swap, and not a one-way message transfer. Of course, the one-way approach is easily mimicked with Rendezvous; a rendezvous on a rv_class of type void, with the matching class having type data, transfers data in only one direction.

2.4.4. Why rv_classes are typed

Rv_classes are typed, in that they allow messages of only one type to be sent, in order to help the programmer restrict and define his application. For instance, if messages are untyped, what value do you send when all you intend is a signal? How does the receiver know that the value sent is a signal, and not a valid datum? Types free the programmer from making such inelegant conversion and convention decisions; a rendezvous on a rv_class of type void acts just as a signal. In addition, the compiler can help the programmer catch conceptual errors by insuring that his program is type-correct.

Typing rv_classes provides a nice protection mechanism as well. Rv_classes are capabilities, and one would like to restrict their propagation through the system. Process A cannot masquerade as process B by taking a message of type data from process B and attempting to rendezvous on it; the kernel will only recognize rv_classes passed as a message under the type rv_class. Furthermore, the system will not allow a caller to pass a
capability that he does not own.

2.4.2. The disjunctive rendezvous

One of the chief complaints about semaphores is that the P(sem) operation does not allow any alternative action if the semaphore is busy. That is, if the value of sem is = 0, the calling process unconditionally blocks. A more general approach is to allow a P to operate on many semaphores:

\[ P( \text{sem}_1 \text{ OR sem}_2 \text{ OR sem}_3 \text{ OR } \ldots ) \]

Our approach is similar. A Rendezvous request specifies a \texttt{ACT} of \texttt{rv_classes}, and means, "Rendezvous on any one of these classes." This causes one's programs to be structured with \texttt{CASE} statements. If there exists a match for more than one \texttt{rv_class} in a given request, the rendezvous succeeds only on the leftmost \texttt{rv_class}. Hence, the calling process blocks only if no other process has requested a rendezvous on \texttt{ANY} of the specified \texttt{rv_classes}.

The choice of a left-to-right examination of triples was motivated by both pragmatic and aesthetic reasons. While there is no compelling philosophical reason to examine the triples in any particular order while searching for a match, fixing the order of the examination allows the user to construct timed and non-blocking rendezvous calls. We will show how to build such extensions after we introduce a few more primitives.

2.4.a. Conditions: Inside or outside the primitive?

A process might want to request a rendezvous on a \texttt{rv_class} only if a certain condition were true. For instance, a bounded buffer manager would
want to rendezvous with a producer to get the next character for a buffer only if the buffer were not full. Similarly, the manager would not want to rendezvous with a consumer unless there were something in his buffer to consume. Consider a bounded buffer manager process that uses a rendezvous primitive without the conditions in the triples (Fig. 2.1). Let us call such a primitive UncondRV. The syntax for the primitive is: UncondRV( < rv_class, msg > )+. Programmed with Rendezvous, with the guards inside the triples, the same program would have only one Rendezvous call (Fig. 2.2). Suppose a process wishes to rendezvous on any one of n rv_classes, under n conditions. Since each condition can be either true or false, \(2^n\) guards, and hence \(2^n\) UncondRV calls, are required to choose the right rendezvous request under the right conditions. However, the Rendezvous primitive conditionally includes triples in a given request, so a single Rendezvous call, and n arms in a case statement, is sufficient.

Buffer_mgr:

\[
\text{if } \text{Bufsize } \rightarrow \text{ "full" } \rightarrow \\
\text{Buf[next_in]} := \text{UncondRV( < producer, \# > )} \\
\text{next_in } :=+ \text{ 1} \\
\text{Bufsize } :=+ \text{ 1} \\
\text{fi}
\]

\[
\text{if } \text{Bufsize } \rightarrow \text{ "empty" } \rightarrow \\
\text{UncondRV( < consumer, Buf[next_out] > )} \\
\text{next_out } :=+ \text{ 1} \\
\text{Bufsize } :=- \text{ 1}
\]

**Figure 2.1.** Bounded Buffer Manager, using UncondRV
Buffer_mgr:
   ...
   ...
   < line, msg, > := Rendezvous( < Bufsize → "full", producer, P >,
       < Bufsize → "empty", consumer, Buf[next_out] >
   )

   case line
     0: Buf[next_in] := msg
        next_in :=+ 1
        Buf_size :=+ 1
     1: next_out :=+ 1
        Buf_size :=- 1
   esac

Figure 2.2. Bounded Buffer Manager, using Rendezvous

2.4.2. Binding time

We have specified that the conditions in the triples are evaluated at
the time of the call. Another possibility for the binding time is to store
the conditions in the rendezvous table, and evaluate the conditions every
time the kernel scans the table to look for a rv_class match. This
interpretation only makes a difference if the calling process blocks and
the conditions involve variables not local to the calling process. Since
the early binding approach is easier to implement than the late binding
approach, and certainly makes the kernel run faster, we have gone with the
early binding time. Furthermore, in our kernel user processes cannot
access nonlocal variables.
2.4.8. Returned values

The triple number value returned by Rendezvous is necessary to allow multiple requests on a rv_class to be included in a single call. If foo is a rv_class, then it is perfectly reasonable to request more than one rendezvous on foo, with a different message for each request:

\[
\text{Rendezvous( } < \text{true, } \text{foo, } \text{msg}_1 >, < \text{true, } \text{foo, } \text{msg}_2 >, \ldots \text{ )}
\]

The triple number return value indicates on which triple the rendezvous actually took place.

The ok return value is negative when an erroneous rendezvous request takes place. If all triple conditions are false at the time of the call, this is flagged as an error. In addition, the kernel traps attempts to use unowned capabilities. We have not specified whether or not a triple condition must evaluate to true for an unowned rv_class appearing in the triple to be caught by the kernel.

It is the compiler's responsibility to catch type errors, as when the type of a message in a triple does not match the type of the triple rv_class.

2.5. Acquiring and releasing rendezvous classes

2.5.1. Syntax
Create( type1, type2 )

Returns: < rv_class1, rv_class2, ok >

Release( rv_class )

Returns: < num_left, ok >

2.5.2. Semantics

The Create primitive will return capabilities for two rendezvous classes with the names rv_class1 and rv_class2. Type1 and type2 specify the message types that can be passed with a rendezvous request with rv_class1 and rv_class2, respectively. A nonzero ok value is returned iff the Create request did not succeed. Possible reasons for failure are: 1) Illegal type specification. 2) Not enough room in this process's capability table.

Rv_class1 and rv_class2 are local names. Thus, in order to rendezvous using those rv_classes, the process making the Create request must pass either or both of the new rv_classes to another process. There are two ways to pass a rv_class:

1) Start up a child process (with the Fork primitive, described below). The child inherits all capabilities owned by the parent.

2) Send the rv_class to another process in a rendezvous message.

Release will remove all copies of the calling process's capability for the specified rv_class, preventing future rendezvous requests on that capability by that process. The matching rv_class is not affected. The num_left value will contain the number of copies of the capability for the released rv_class owned by all other processes plus the number of copies of
the capability for the matching rv_class extant. If the calling process
does not own the specified rv_class, then the ok value is < 0.

2.5.3. Dynamic Create and Release

Dynamic creation and destruction of rendezvous capabilities, besides
being of obvious importance in the restriction of rv_class scope, allow
processes to establish private communication lines. Consider a system
resource manager process. User processes make resource requests by per-
forming a rendezvous on the globally known (i.e., owned by every process)
rv_class REQ_RES. REQ_RES is of type rv_class, and matches the rv_class
REQ_RES_MATCH. The resource manager creates rendezvous classes to conduct
a private conversation (Fig. 2.3). A user process is passed the
Get_resource capability, on which he should rendezvous to actually acquire
the resource he requested with REQ_RES. The Get_resource and Give_resource
rv_classes created on different loop iterations are different capabilities.
A malicious user cannot steal resources by requesting many rendezvous calls
on Get_resource, since the resource manager releases Give_resource after
the first rendezvous on it. (A malicious user can, however, permanently
block the resource manager by refusing to rendezvous even once on
Get_resource. We will deal with non-blocking rendezvous below.)

2.5.4. Keeping track of rv_classes

Although there are four ways to acquire rv_classes (through the
Create, Rendezvous, Assoc, and Fork primitives) an active process can
relinquish rv_classes only through Release or Exit. The num_left value
returned by Release gives the sum of the number of capabilities in the sys-
tem for both the specified rv_class and its match. A process might wish to
Resource_mgr:

do forever +
  < Get_resource, Give_resource, > := Create( void, data )
  < , msg, > := Rendezvous( < TRUE, REQ_RES_MATCH, Get_resource > )
  Examine msg, and construct response datum.
  Rendezvous( < TRUE, Give_resource, response > )
  Release( Get_resource )
  Release( Give_resource )
odo

Figure 2.1. Establishing a Private Communication Line

investigate the situation if it finds more or less copies of a rv_class capability extant than it expects or desires.

We have said that capabilities are not rv_classes. The Create primitive will actually cause the kernel to create a new pair of rv_classes; the caller is given a capability to each of the new classes. A process can acquire multiple capabilities to the same rv_class by being passed the capabilities as a Rendezvous message, and Release works in the logical way: It removes only the specified capability, not all capabilities that map to the same rv_class.

2.6. Spawning a child process

2.6.1. Syntax
Fork()

Returns: who_am_I

Exit()

2.6.2. Semantics

The Fork primitive will create a new copy (a child) of the calling process (the parent). The child will execute concurrently with the other system programs. Execution of the child begins at the point immediately after the Fork() call.

The child is an exact copy of the parent (including the parent's memory, environment, capabilities, etc.), with two exceptions:

1) The child has different memory management registers, since the child has its own memory space. The child's memory is an exact copy of the parent's.

2) The value of who_am_I returned to the parent is different from the value returned to the child.

A call to the Exit() primitive kills the calling process; that is, it terminates its execution, releases its capabilities, and removes it from the process table.

2.6.3. How to spawn

It was tempting to include a primitive to implement strict separation between parent and child. Such a primitive might have the syntax Spawn(<process_name, {rv_class}+>), meaning, "Start up a new instance of the procedure starting at process_name, and give it copies of the following
rv_classes." The first statement at process_name would be of the form Inherit(< rv_class >), in order to give local names to the rv_classes inherited from the parent. Fork is a lower-level primitive, in that one can only implement Spawn in terms of Fork (Fig. 2.4). Since we are trying to design an operating system kernel, we feel that our primitives should really be primitive. The next level of an operating system based on the Rendezvous kernel could include a Spawn built from Fork. Hence, we have not arbitrarily restricted the power of the primitives by specifying that only part of the parent's environment is inherited by the child.

\[
\text{Spawn( kid, cl, c2 ) } \iff \begin{align*}
\text{who}_\text{am}_i &:= \text{Fork()} \\
&\text{if } \text{who}_\text{am}_i = \text{CHILD} + \\
&\quad \text{Release( all but cl and c2 )} \\
&\quad \text{goto kidcode} \\
&\quad \text{if } \text{who}_\text{am}_i = \text{PARENT} + \text{skip} \\
&\text{fi}
\end{align*}
\]

Figure 2.4. The Spawn Operation

The term "clone" would be more appropriate than "parent" or "child" in reference to the result of a Fork call, since the "parent" and "child" terms tend to imply a hierarchical relationship that really does not exist. The parent is not special in any way to the child after a Fork; if the parent dies with Exit, the parent's children do not die too.

2.7. TREATING AN INTERRUPT AS A RENDEZVOUS

2.7.1. Syntax
Assoc(vector address)

Returns: <wfi_rvc, ok>

2.7.2. Semantics

The Assoc primitive will create a new rendezvous class pair. One member of the pair is the rendezvous class wfi_rvc. The other member of the pair is associated with the specified interrupt vector address. After the Assoc call is performed, an interrupt at the vector address will generate a rendezvous that matches wfi_rvc. Hence, in order to wait for an interrupt from the device corresponding to the vector address, a process must request a rendezvous on wfi_rvc.

The message type of wfi_rvc is void.

Only one Assoc on a given vector address is allowed in the entire system.

The ok return value will be nonzero iff the Assoc call did not succeed (illegal vector address, no space in the calling process's capability table, an Assoc on the same vector address has already been performed).

2.7.3. Interrupt handlers

The Assoc primitive subsumes interrupts into the lowest level of the system, allowing interrupt handlers to be written that look basically like any other server programs. Device communication is not very different from interprocess communication. Suppose we have an interrupt routine to handle characters to and from a terminal (tty). The program fragment in Figure 2.5 sends incoming characters to an editor program (outgoing characters are handled in a similar fashion).
<Wait_for_interrupt> := Assoc(tty_addr)
do forever +
   <line, msg> := Rendezvous(<true, Wait_for_interrupt, X>);
   Bufsize ← "empty",
   To_editor, Buf[next_out] >
   
   case line
   0: /* Interrupt; new character available */
      Buf[next_in] := device.data_reg
      next_in :=+ 1
      Bufsize :=+ 1
   1: /* Character sent from tty to editor */
      next_out :=+ 1
      Bufsize :=- 1
   esac
od

Figure 2.5. An Interrupt Handler Prototype

2.8. Timed rendezvous

It may be desirable to have a rendezvous call that only delays a finite amount of time if no rendezvous requests succeed. A clock manager process that implements a timer can be built using the above primitives. An example of the use of the timer is given in Figure 2.6. A rendezvous on the wakeup triple in the second call will occur iff no rendezvous with any other triple in the call occurred in 17 ticks. This is exactly the semantics that one would expect of a timed rendezvous call.

We have shown the timed rendezvous as a sequence of two statements, but if we prefer to program applicatively, we could, with some syntactic finagling, collapse the first call into the last triple position in the
< , wakeup > := Rendezvous( < true, Clock_mgr, 17 > )
/* A rendezvous on the rv_class matching wakeup */
/* will occur after 17 ticks. */
< line, msg, > := Rendezvous( . . . < true, wakeup, weekday > )

Figure 2.6. Timer Usage

second call. A powerful macro processor would probably be needed.

2.8.1. Non-blocking rendezvous

It may not be desirable to have the calling process block if all triples in a call fail to find a match. If the last triple in a call is a "wakeup after 0 tick" triple, then the triple has the desired semantics: "Rendezvous with me iff no other triple has an immediate match in the rendezvous table." Note that the programmer must be careful when using the non-blocking rendezvous. If the processes at both ends of a message exchange decide to use the non-blocking rendezvous, then race conditions open the possibility that no message exchange will ever take place. Presumably each process is awakened almost immediately, so its triples do not remain in the rendezvous table very long.

2.8.2. The clock manager

We will give the code for the clock manager in full, as it illustrates an important design technique. We mentioned above that a resource manager might be blocked indefinitely if a malicious user process does not rendezvous on the "private line" rv_class it received from the manager. The technique for dealing with this is very simple. The manager, after sending

*
a user process the private line rv_class, spawns a child, giving the child the matching private line rv_class. The child attempts to rendezvous with the user and give the user the requested resource. The child may block forever if the user turns out to be malicious, but the parent resource manager will not.

Here is an overview of the operation of the clock manager:

1) User FOO performs a rendezvous on the globally known Timer rv_class, passing "wake me up in 17 clock ticks" as the datum. FOO receives in return a Wakeup_call rv_class.

   +---------+  Wakeup_call  +--------+
   | clock   |<---------------------| FOO   |
   | manager |<---------------------|       |
   +---------+                 +--------+

   Figure 2.7a. Clock Manager: Initial Exchange

2) The clock manager spawns child.FOO, giving it the rv_class Alarm, which matches Wakeup_call. Meanwhile, FOO has requested a rendezvous on Wakeup_call.

3) On every hardware clock tick, the clock manager sends a software clock tick to all its children.

4) After child.FOO has received 17 software ticks, it wakes up FOO, and commits suicide.

Finally, Figure 2.8 presents the clock manager process itself.
2.9. Setting a process stack size
2.9.1. Syntax

Stksize( n )

Returns: ok

2.9.2. Semantics

The Stksize primitive will grow or shrink the size of the stack area of the current process to be at least n bytes. The return value ok flags an error if the requested stack area size would be smaller than the current stack (as indicated by the process stack pointer).

Stksize copies the current stack if the requested stack area is larger than the current stack area. The kernel sets the calling process's stack pointer to reflect the new stack location.

2.9.3. Frugal memory organization

With Stksize, user code can be written to allow a new process to copy part of its stack to the high end of the stack area, and then shrink the stack area. This lets a child process throw away the stack generated by its parent, keeping only that part of the parent's stack essential to the child.

2.10. Setting a process priority

2.10.1. Syntax

Setpri( p )

Returns: ok
2.10.2. Semantics

The Setpri primitive will set the calling process's scheduling priority to p. The identifier ok flags an illegal request.

2.10.3. Scheduling

We have given a minimum scheduling specification. All that the priorities indicate is that if two processes are waiting to run, the one with the higher priority will be run first. Processes with equal priority are scheduled in round-robin order.

Since a child process inherits its parent's environment, a child will inherit its parent's priority. The root user process is assumed to be running at the lowest priority until it makes an explicit Setpri call.
Clock_mgr: /* Assume that timeoutreq_rc matches the globally known user timer request class. */

< htick, > := Assoc( clock_addr ) /* Clock interrupt (hardware */
// tick) */

Integer nchildren /* Number of clock mgr children */
Integer flip /* Software tick polarity */

/* Software ticks are sent by the clock mgr to its children after each hardware tick. The manager will send nchildren software ticks on stick_p[0], and send the next software tick on stick_p[1]. The flip variable alternates between 0 and 1. The alternation is necessary, since if child A attempts to rendezvous on the same rv_class from which it just received a software tick, the parent may not have finished distributing the first software ticks to its other children. Thus, child A may see the same hardware tick as two software ticks. */

< stick_p[0], stick_c[0] > := Create( void, data )
< stick_p[1], stick_c[1] > := Create( void, data )

< Wakeup_call, Alarm, > := Create( void, void )

nchildren := 0
flip := 0
do forever +

< line, msg, > := Rendezvous( < true, htick, ₱ > )

< true, timeoutreq_rc,
   Wakeup_call >

)

case line

0: /* Hardware tick */
do ∀ children +

< * msg, > := Rendezvous( < true,
   stick_p[flip], ₱ > )
if msg = "I'm dying" → nchildren := - 1
[] msg = "I'm alive" + skip
fi

od
flip := 1 - flip

Figure 2.2. The Clock Manager
1: /* Timeout request */
   who_am_i := Fork()
   if who_am_i = CHILD +
   /* Clock mgr child proper */
   Time_until_done := msg
   do Time_until_done > 0 +
      Rendezvous( < true, 
                  stick_c[flip],
                  \"I'm alive\" > )
      flip := 1 - flip
      Time_until_done :=- 1
   od
   /* Signal to clock mgr
      that I'm done,
      wake up user, and die
   */
   Rendezvous( < true, 
               stick_c[flip],
               \"I'm dying\" > )
   Rendezvous( < true, 
               Alarm, X > )
   Exit()

□ who_am_i = PARENT +
   Release( Wakeup_call )
   Release( Alarm )
   < Wakeup_call, Alarm > := Create( void, 
                              void )

fi

done

Figure 2.8. (continued)
Chapter 3

Sample Applications: The Crossbar Switch

1.1. *Overview*

In this chapter we present the Crossbar Switch, an example of the use to which the rendezvous primitives might be put. We first motivate the Crossbar Switch by discussing the limitations of conventional device/backend machine configurations. We then present the idea of a Crossbar Switch, a generalized virtual connection and reconfiguration system. After proposing a model for the Crossbar Switch, we show how the Rendezvous primitives can be used to implement the model.

1.2. *Drawbacks to standard configurations*

The standard configuration between a terminal (tty) and a backend machine (bm) allows the terminal users and the backend machine processes very little flexibility. If we ignore the details of the actual interface (DZ-11 terminal multiplexors, etc.), then there is a 1-1 correspondence between tty's and bm ports (Fig. 3.1). A terminal is permanently attached to a single bm port unless someone physically reconfigures the system by plugging and unplugging connection wires. A more general scheme would allow a terminal to be hooked into arbitrarily many ports at once, with the user switching back and forth between ports at will. In essence, such a scheme creates virtual terminals. By having his terminal connected to n ports at once, a user can pretend he is surrounded by n physical terminals.
each hooked into a single port.

```
| backend machine |
+---|---|---|
|    |   |   |
+---+---+---
|    |   |   |
+---+---+---
terminals (tty's)
```

Figure 3.1. Conventional backend machine/tty configuration

If more than one backend machine is available, we would like to give a terminal even greater reconfiguration power: A terminal should be able to be hooked into arbitrarily many ports on any available backend machine.

In addition, standard configurations do not deal gracefully with unusual hardware or unusual applications. For example, a modem poses configuration problems, because it is used sometimes as a dial-in port (when a remote user dials up the backend machine), and sometimes as a dial-out port (when the backend machine dials another computer). An example of a problematic application is a bm to bm file transfer. In standard configurations, one would have to dedicate a port on each machine to handle nothing but file transfers. A flexible system, in which "virtual configurations" can be established and broken at will, is much more desirable. In such a system, even rather fanciful applications (e.g., a direct terminal-to-terminal connection) should be simple to set up.
3.2. The Crossbar Switch approach

We define a Crossbar Switch (CS) as a computer dedicated to the support of virtual configurations. That is, the Crossbar Switch provides a means for connecting together devices in an arbitrary fashion. Connections are not physical, but are supported by software. Once connected, devices communicate by sending and receiving characters through CS ports.

A broad-brush view of a typical Crossbar Switch is given in Figure 3.2. In this scenario, two backend machines, A and B, are physically connected to the CS, as are two tty's and a modem. The CS is the center of the system; no device or backend machine is physically connected to anything but the Crossbar Switch. We define a CS port as an entry point into the CS. A connection is a virtual line over which devices pass characters. The function of the CS is to provide a connection between any two devices that want to talk to each other. A connection lasts only until the devices request that the connection be broken.

Figure 3.2 illustrates that a virtual terminal facility is supported by the CS. In the picture, port a is connected to port h, port b is connected to port k, and so forth. A physical terminal, tt1, has four lines into the CS, at ports j through m; another terminal, tt2, has two lines, at ports n and o. (Note that the multiplexing of a physical tty line into many virtual tty lines takes place outside the CS proper. We will return to this point later.) tt1 has a connection to backend machine B, and has two separate connections to backend machine A; its fourth port into the CS, port 1, is not connected to anything at the moment. tt1 is currently

---

1This is not the most reliable configuration in the world, since if the CS goes down, everything goes down. We have purchased flexibility at the price of reliability.
using the connection at port k.

tty2 is connected to backend machine A and a modem, and is currently using the modem connection (in, presumably, a dial-in capacity). There is no reason why the modem could not, when its current conversation is over, be connected to a backend machine for dial-out purposes.

The backend machines are connected through CS ports a and b. Such a connection would be useful for direct file transfers.
3.4. The CS/device Interface

Before delving into the proposed model for the internal operation of the CS, it is necessary to consider how devices and backend machines would interface into the CS, and how devices would do "intelligent" things like multiplexing and requesting connections.

Since our Rendezvous kernel is implemented on a DEC PDP 11/60, consider a CS based on that machine. The CS is one of many logical constructs living within the 11/60 (Fig. 3.3). The CS ports are logical ports; the boxes drawn outside the border of the 11/60 are physical ports. The basic idea is that every device using the CS must have some intelligence. A backend machine clearly can set up processes to talk to the CS, and request that connections be made and broken. When a device with little or no processing power must use the CS, then a smart emulator for that device must run on the 11/60, interfacing the device to the CS. Thus, the smart terminal emulator (STE) handles the task of creating a virtual terminal by multiplexing a single physical tty line into many logical lines into the CS. The CS does not verify the identity of those requesting connections; it merely establishes connections between two CS ports. The STE performs the requisite security operations, such as handshaking across a connection.

3.5. The CS model

In this section we describe the CS model, including a discussion of the high-level connect/disconnect protocols. We also describe the CS internal processes and how they interrelate. The smart emulators create virtual smart devices for the CS, so in the rest of this chapter we assume that the devices using the CS really do have some processing power. In
fact, we can abstract away the 11/60 interface altogether, and assume that the CS is a physical machine connected to physical devices.

3.5.1. Connections and connection classes

As we have said, a CS port is an entry point into the CS. A CS port driver process (port process, for short) handles the flow of characters into and out of a port. External devices form connections through the CS in order to communicate. The overall structure of the CS is given in Figure 3.4. External devices X and Y have established a connection. Device X is connected through port a, handled by port process A, and device Y is connected through port b, handled by port process B. Ports c, d, and e are currently unused.

Connections are partitioned into classes. A connection class describes a service sought by one port process and provided by another.
Figure 3.4. A Connection Across the CS

For instance, a login connection would be sought by a tty, and provided by a backend machine login process. The ends of the connection are, then, not symmetric; they can be described variously as provider-seeker, producer-consumer, male-female, and so forth. We will use the male-female descriptors. Thus, each of 2 connected ports will have a gender (male or female) for that connection class.

We will discuss the connection protocol in detail below, but a brief summary of how a port process views a connection might be helpful for now:

1) An external device requests a connection through the CS.

2) The port process acquires a set of Rendezvous capabilities.

3) The port process uses its Rendezvous capabilities to send and receive characters over the connection.

4) The external device requests that the connection be broken.
5) The port process releases its Rendezvous capabilities.

3.5.2. Division of labor

A port process receives and transmits characters to and from its external device; it knows little or nothing of character semantics. Each port process has a manager process that performs the initial task of assembling the characters from the port process into a connection request. Thus, for each port there is one port process and one manager process.

The CS connection manager handles all connection requests from the manager processes. The function of the connection manager is to act as a matchmaker, connecting up port processes that have requested the same connection class but opposite genders. A pool of connection processes remain inactive until activated by the connection manager. The situation after one port process has requested a connection is shown in Figure 3.5. After a connection has been established, all characters are passed through a connection process. The connection manager and the port managers are no longer involved (Fig. 3.6).

![Figure 3.5. Connection Manager](image-url)
3.6. Operation of the CS

3.6.1. The connection protocol

1) A port process (call it port A) passes incoming characters to its manager process. The manager assembles the characters into a connection request, and performs a rendezvous with the CS connection manager on a "connection requested" rv_class. The message passed to the connection manager consists of a set of connection classes and a gender for each class. Such a message is called a connection request.

2) Like the clock manager discussed in Chapter 2, the connection manager is a server; its function is to service connection requests. The connection manager will, like the clock manager, establish a private line with its clients. Hence, the connection manager will continually rendezvous on the match for the "connection requested" rv_class, and will eventually receive port A's connection request.

3) When the rendezvous occurs, port manager A will receive a "accept new capabilities" rv_class from the connection manager. The "accept new capabilities" rv_class is unique to this connection request. Port manager A performs a rendezvous on the "accept new capabilities" rv_class. Port manager A and the connection manager have now established a private Rendezvous line.
4) The connection manager keeps a list of outstanding connection requests in the **connection table**. The connection manager compares port A's connection request with the entries in the connection table. If no match is found, then port A's connection request is added to the table.\(^2\)

5) If the connection manager does find a match (with, say, port B) for port A's connection request, it will create rendezvous classes for the connection. (The port processes will eventually use these rv_classes to send characters to and from the connection process.) The connection manager will then rendezvous with a idle connection process, passing it the new rv_classes.

6) The connection manager performs a rendezvous with port managers A and B (using the appropriate "accept new capabilities" rv_classes), and passes to each port manager the identity of its connection partner, and the rv_classes its port needs to rendezvous with the connection process.

7) Each port manager passes the new connection rv_classes to its port process. The port processes transfer all further characters by invoking Rendezvous with those rv_classes.

8) The connection manager and the port managers play no further part in the connection. Each port process sends and receives characters by performing a rendezvous with the connection process.

\(^2\)Note that establishing a connection is beginning to bear a strong resemblance to requesting a rendezvous. We have deliberately chosen a different method of establishing connections (with connection classes and genders, instead of matching connection classes) in order to keep the distinction clear.
At any time a port is either connected to another port or it is disconnected. Disconnect is discussed in detail below.

1.6.2. Port process/Connection process interaction

When a connection starts up, each port process receives three rv_classes: "to CP", "from CP", and "disconnect msgs". The connection process receives six rv_classes, the matching classes for each of the port process sets. When a port process has a character to send across the connection, it uses the "to CP" rv_class, with the character to be sent as the rendezvous message. Similarly, to receive a character, the port process uses the "from CP" rv_class. The returned message is a character from the other end of the connection.

Rendezvous effects a message swap, so it may seem that we could get by with one data-transfer rv_class between the port processes and the connection process: The message passed is a character destined for the device at the other end of the connection, and the message received is a character from the device at the other end. However, the connection process has a bounded buffer, and should not request a rendezvous with the match for "to CP" if its input buffer for that port process is full. Likewise, the connection process should not request a rendezvous with the match for "from CP" if it has nothing to give that port process (i.e., it its output buffer is empty). So two rv_classes, one to pass data in each direction, are necessary.

The "disconnect msgs" rv_class is used by the port process to request that it be disconnected, and is used by the connection process to signal to a port process that the other member of the connection has disconnected.
1.6.3. Disconnect protocol

Suppose a connection process (CP), handling ports A and B, receives a disconnect request from port process A. The CP will release its connection capabilities for port process A, and no longer attempt to rendezvous with B's "to CP" rv_class. Since the CP is buffering characters between A and B, B should be able to read buffered characters from A even after A has actually disconnected. However, once the A-to-B buffer is empty, the CP will no longer rendezvous on B's "from CP" rv_class. Instead, the CP will signal through the "disconnect msgs" rv_class that A has disconnected.

Connection requests are distributed, in that both ends must request a connection in order for the connection to be established. In the interests of symmetry we require that disconnect requests be distributed also. Hence, the CP will now rendezvous on B's "disconnect msgs" rv_class again, waiting for B's "disconnect me" message. Once the CP receives B's disconnect request, it will release B's connection rv_classes, and rendezvous with the connection manager using the "Free connection process" rv_class.

3.7. CS code

The high-level program descriptions of the Connection Manager, Connection Process, Port Manager Process, and Port Process are given in Figures 3.7.

2.8. Alternative designs

As the reader might suspect, we have based our Crossbar Switch design more upon ease of explication than quality of the final product. For example, the separation of the port process and manager process functions is
quite unnecessary for other than didactic purposes, and the two processes can be folded into one. The separation of function can be preserved by converting the manager process into a manager procedure. In that case, the port process passes data to the port manager by calling it as a procedure instead of through Rendezvous.

There need not be a pool of blocked connection processes standing idle until called into action by the connection manager; a new connection process can be started up when needed with Fork. The connection process would kill itself with Exit when the connection is over.

Since one port and one manager process exist for each CS port, folding them into a single process is a reasonable thing to do. What is perhaps not so obvious is that the connection processes can also be done away with. In the implementation described above, the CP simply manages a buffer pair. There is no reason that each port process cannot handle a buffer pair on its own end, and directly communicate across a connection with another port process. In fact, a buffer pair is also unnecessary; each PP could handle a single buffer at its own end, buffering characters received from the other PP. Characters received from the port device would be sent directly to the other PP using Rendezvous. Our only objection to these and similar approaches is that they make the PP code slightly more complicated than the straightforward code in Figure 3.7d, particularly when the elimination of the CP's is combined with the folding of the manager process into the port process.
Conn_Mgr: /* Our Founder */

/*/ Spawn Connection Processes */

< Wakeup_CP, Free_CP, > := Create( void, rv_class )
do ∀ CP's +
  who_am_i := Fork()
  if who_am_i = CHILD +
    Release( Wakeup_CP )
    I'm_free := Free_CP
    goto CP
  fi
  who_am_i = PARENT +
  skip
od
Release( Free_CP )

/*/ Spawn Port Mgr Proc */

< Accept_conn_req, Make_conn_req, > := Create( void, data )
do ∀ Port Mgrs +
  who_am_i := Fork()
  if who_am_i = CHILD +
    Release( Accept_conn_req )
    Conn_req := Make_conn_req
    goto Port_mgr
  fi
  who_am_i = PARENT +
  skip
od

Initialize connection table.
do forever +

/*
 Get new connection request. Rendezvous on the general request line first, then establish a private line with the Port Manager requesting the connection.
*/

< Send_caps, Receive_caps, > := Create( rv_class, void )
< Receive_rest_of_req, Send_rest_of_req, > := Create( void, data )
Rendezvous( < true, Accept_conn_req, Receive_caps > )
Release( Receive_caps )
Rendezvous( < false, Send_caps, Send_rest_of_req > )
Release( Send_rest_of_req )
msg = "none yet"
do msg := "end of request" +
  < , msg, > := Rendezvous( < true, Receive_rest_of_req, P > )
  Add msg to connection request
od
Release( Receive_rest_of_req )

Search connection table for a match.

Figure 3.7a. Connection Manager
if found +

/* Assume the connection is between mgr procs j and k.
   Generate 6 pairs of rv_classes for the new connection:

   +-----+    +-----+    +-----+
   | data  |
   |       |
   +-------+    +-------+    +-------+
   | data  |
   |       |
   PP    CP    PP
   j      k

   "disconn. msg"
   "disconn. msg"

   +-----+    +-----+    +-----+

   The arrows indicate the direction in which messages
   are passed.
*/

/* Give capabilities to PPj */
< c[1], c[2], > := Create( void, data )
< c[3], c[4], > := Create( data, void )
< c[5], c[6], > := Create( data, data )
do i = 1 to 5 by 2 +
   Rendezvous( < true, Send_capsj, c[i] > )
od
Release( Send_capsj )

/* Give capabilities to PPk */
< c[7], c[8], > := Create( void, data )
< c[9], c[10], > := Create( data, void )
< c[11], c[12], > := Create( data, data )
do i = 7 to 11 by 2 +
   Rendezvous( < true, Send_capsk, c[i] > )
od
Release( Send_capsk )

/* Give capabilities to CP */
<, Fresh_CpP > := Rendezvous( < true, Wakeup_CP, P > )
do i = 2 to 12 by 2 +
   Rendezvous( < true, Fresh_CP, c[i] > )
od
Release( Fresh_CP )

/* Release the created caps */
do i = 1 to 12 by 1 +
   Release( c[i] )
od

Figure 1.7a. (continued)
ilocally -> found ->

Enter connection request in connection table. Store the Send_caps rv_class for this request along with the connection request.

Figure 3.7a. (continued)
Figure 1.7b. Connection Process
0: Buf2[next_in] := msg
   next_in :=+ 1
   size :=+ 1

1: next_out :=+ 1
   size :=- 1

2, 3: if msg = "disconnect me"
    then A_conn := false
    do i = 1 to 3 + Release(c[i]) od
    otherwise + skip
   fi

   .
   .
   . (Cases for B are analogous)
   .
   .

end

Figure 3.7b. (continued)
Port_mgr: /* Inherited rv_class: Conn_req */

/* The rendezvous messages between the Mgr and its PP consist of 
   two data lines and one line for the acquisition of new rv_classes:

     +-----+     +-----+
     | data |     | data  |
     |      |  +-------+-------+  |
     | PP   |     | Mgr   |
     | <-----|     |        |
     | new conn. caps |        |
     |        |     +-------+-------+  |
     +-------+     +-----+

*/

/* Create the req'd caps, spawn the PP, and give the caps to the PP */
< c[1], PP_in, > := Create( data, void )
< c[2], PP_out, > := Create( void, data )
< c[3], New_caps, > := Create( void, rv_class )
who_am_i := Fork() +
if who_am_i = CHILD +
   Release( PP_in )
   Release( PP_out )
   Release( New_caps )
   Release( Conn_req )
   goto Port_proc
fi
if who_am_i = PARENT +
   skip Port_proc
fi
Release( c[1] )
Release( c[2] )
Release( c[3] )

do forever +

/* Get new connection request from device. Note that the connection 
   request might be hard-coded for some devices. Some mgr_procs 
   might send chars to the device (using the PP_out rv_class), 
   but we have not included that code in this mgr_proc prototype. */

Conn_request := "none yet"
do until Conn_request is assembled +
   < , msg, > := Rendezvous( < true, PP_in, \> )
   Add msg to Conn_request
od

/* Establish a private line with the connection manager, and 
   submit the connection request. Assume the request is in 
   the form Conn_request[1:n]. */

< , Get_caps, > := Rendezvous( < true, Conn_req, \> )

Figure 3.7c. Port Manager Process
< , Send_req, > := Rendezvous( < true, Get_caps, x > )
\[\text{do} \ i = 1 \ \text{to} \ n \ + \]
\[\text{Rendezvous( } < \text{true, Send_req, Conn_request[i], } > \text{ )} \]
\[\text{od} \]

/* Get new connection capabilities */
\[\text{do} \ i = 1 \ \text{to} \ 3 \ + \]
\[\text{< , c[i], > := Rendezvous( } < \text{true, Get_caps, x, } > \text{ )} \]
\[\text{od} \]

/* Give the caps to PP */
\[\text{do} \ i = 1 \ \text{to} \ 3 \ + \]
\[\text{Rendezvous( } < \text{true, New_caps, c[i], } > \text{ )} \]
\[\text{od} \]

\[\text{Figure } 3.7c. \ \text{(continued)}\]
Port_proc: /* Inherited rv_classes: Mgr_out, Mgr_in, New_caps */
/* PP has 4 permanent capabilities (used to talk to the Mgr and the
device) and 3 transitory capabilities (used to talk to a CP):

* /
/* Set up device rv_class for the device using interrupt vector 250 */
<Wait_for_int, > := Assoc( 250 )
Disc := true /* true if PP is disconnected */
do forever +
    device.data_reg := value /* PDP 11/60 has memory-mapped
    status driven IO */
    device.status_reg := GO
/* Interrupt => new byte is avail. */
    Rendezvous( true, Wait_for_int, 1 > )
    stuff := device.data_reg
    if (stuff = 'disconnect me' control character) & (~Disc) +
        Disc := true
        Rendezvous( <true, CP_disc, "disconnect me" > )
        Release( CP_out )
        Release( CP_in )
        Release( CP_disc )
    [] otherwise + skip
    fi
< line, msg, > := Rendezvous( <~ Disc, CP_out, stuff >,
                                     < Disc, CP_in, 1 >,
                                     <~ Disc, CP_disc, 1 >)

Figure 3.7d. Port Process
< Disc, Mgr_out, stuff >.
< Disc, Mgr_in, _ >.
< Disc, New_caps, _ >
)

case line

0: value := _
1: value := msg
2: value := "The other end has disconnected"
3: value := _
4: value := msg
5: Disc := false
   CP_in := msg
   < , CP_out, > := Rendezvous( < true, New_caps, _ > )
   < , CP_disc, > := Rendezvous( < true, New_caps, _ > )
   value := _

esac

od

Figure 1.7d. (continued)
Chapter 4

Implementation of the Rendezvous Kernel

4.1. Overview

This chapter describes our implementation of the rendezvous primitives on a DEC PDP 11/60. The implementation comprises a rendezvous kernel on which user programs can be run or the next level of an operating system built. After describing the kernel interface to the C programming language, we describe the data structures, memory management, process switching, and sundry hardware-specific details. After giving a brief overview of the organization of the kernel programs, we conclude with some suggestions for improving the performance, security, and flexibility of the kernel in future implementations.

This chapter assumes that the reader is familiar with the C programming language, the PDP 11/60 hardware and programming paradigms, and the UNIX-style interface between C and the 11/60 machine language.

4.2. Interface to C

In Chapter 2, when describing the semantics of the rendezvous primitives, we used meta-notation to describe the number of values expected and returned by a primitive. This section describes the implementation of the meta-notation; that is, how arguments are passed and errors are encoded in our C implementation. We will discuss each of the primitives in detail
All primitives have the same general syntax as function calls. The call syntax for Create is:

\[ \text{ok} = \text{create} ( \&\text{cap1}, \text{TYPE1}, \&\text{cap2}, \text{TYPE2} ); \]

Note that the C names for the primitives are not capitalized. The return locations for the created capabilities are specified in the argument list, along with the desired message type for each capability. The kernel provides a set of macros (in UPPERCASE letters) that the user must use to specify types. There are currently three types available: \text{T\_DATA}, \text{T\_RVCLASS}, and \text{T\_VOID}. The return value of the Create, which in the above example is assigned to the identifier \text{ok}, is \text{OK ( = 0 )} if the Create was successful; in this case, the new capabilities are in \text{cap1} and \text{cap2}. If the Create was not successful, then Create returns a negative value. The user can determine exactly what went wrong by comparing \text{ok} with the system-provided error macros. For Create, there are three possible error conditions:

\text{E\_BADTYPE}. One or both of the user-specified types is not a legal type.

\text{E\_PRVC}. The user has no room in his private rendezvous class table. (This will become clearer when we discuss the user process data structures, below.)
E_R_SRVC. The system has no more rendezvous classes to give out (they are a finite resource).

As can be seen from the above examples, system macros are generally typed with their prefix. That is, a macro describing rendezvous class types begins with T_; a macro describing an error condition begins with E_; and a macro describing an error condition resulting from a resource limitation begins with E_R_. All macros beginning with E_ denote negative values. The values of all other macros are nonnegative.

If one were to use the primitives in a strongly typed language in which it is possible to define types, then the return location identifiers cap1 and cap2 should be typed as rv_class. In C, rv_class capabilities are denoted by small integers, ranging from 0 to NPRVC-1 (NPRVC is a macro denoting the Number of Private RendezVous Capabilities for each user process). The kernel provides a C type-definition construct to allow the user to declare rv_class identifiers, viz. "typedef int rv_class;".

4.2.2. Release

The call syntax for Release is:

\[\text{num_left} = \text{release( cap )};\]

where cap and num_left are integers. The return value is E_BADCAPY if the user does not own the argument capability. Otherwise, the return value is the sum of the number of capabilities to the rv_class denoted by cap and its matching rv_class extant in the system.
4.2.3. Rendezvous

The abstract notation for a rendezvous call allows the argument list to consist of one or more triples. In practice, a triple is denoted by three values separated by commas. The first value in the argument list is the number of triples in the call; the rest of the values are the triples themselves. For instance, the abstract rendezvous call

\[
< \text{tripnum}, \text{rmag}, \text{ok} > := \text{Rendezvous}( < \text{true}, \text{cl}, \text{msg1} >, \\
< \text{a & b}, \text{c2}, \text{msg2} > )
\]

would be written in C as:

```c
tripnum = rendezvous( 2, TRUE, cl, msg1, (a & b), c2, msg2 );
```

The initial 2 in the argument list indicates that two triples follow. TRUE is a macro that evaluates to `true`; there is an analogous FALSE macro. If nonnegative, the return value of the call indicates on which triple the rendezvous succeeded. The return value is assigned to the identifier tripnum above. If the return value is negative, then a rendezvous did not take place. The following macros describe possible rendezvous failures:

- \texttt{E\_TRIPCOUNT}. The triple count in the first argument is invalid. There is no limit on the number of triples in the argument list, but the kernel will check to insure that the user's stack can actually contain the specified number of triples.

- \texttt{E\_RADCAPX}. The capability specified in the \texttt{rv\_class} field of one or more triples is not owned by the caller. The user is given no indica-
tion *which* triple contains the invalid *rv_class* field. The error handling facilities of the kernel are not very sophisticated at present, and this variety of minimally informative error message is the norm in all the primitives. Future versions of the kernel should include more extensive error reporting facilities.

**E BADMSGCAPY.** For one of the triples with a *rv_class* that passes messages of type *rv_class*, the message field does not contain a valid capability.

**E RPRVC.** At least one of the triple *rv_classes* has a matching *rv_class* of type *rv_class*, meaning that the caller will receive a capability in return if a rendezvous succeeds on that triple. However, the caller has no room in his private capability table for a new capability.

**E ALLCONDSFALSE.** The condition field of all triples in the argument list have evaluated to false, so there are no triples with which to make a rendezvous request.

The rendezvous return message itself is returned in the unsigned variable *returnmsg*. Every user process has its own copy of *returnmsg*, which it declares with an "extern unsigned *returnmsg*;" statement. Unless the rendezvous call returns with *tripnum < 0*, or the return message type is *void*, the value of *returnmsg* changes with every rendezvous call. Of course, the user is free to copy or change the value of *returnmsg* at any time; aside from being set by a Rendezvous call, *returnmsg* is just like any other local user variable.
4.2.4. **Fork**

The Fork primitive is called with no arguments: `which = fork()`. The return value is PARENT for the original calling process, and CHILD for the newly wrouth clone process. There are only two error conditions. E_R_NOMEMLEFT indicates that the child was not created, because there is not enough physical memory to allocate to its stack. E_R_PENODES indicates that the kernel process data structure does not have enough space for a new process entry.

4.2.5. **Exit**

Exit is also called with no arguments: `exit()`. Exit always succeeds, and there is obviously no return value.

4.2.6. **Assoc**

Assoc is called in the conventional fashion:

```
cap = assoc( vector_addr );
```

If the return value is nonnegative, then it is a new capability. If the return value is negative, then the Assoc request did not succeed. The error values E_R_PRVC and E_R_SRVC have the same meaning as described in the section on Create, above. E_R_ATAB means that the kernel has no room in its device association table to enter the caller's association request. E_BADVEC indicates that the caller has specified a nonexistent or illegal trap vector address, while E_VECINUSE means that some user has previously associated that vector with a device. (Recall that only one association per vector is allowed throughout the entire system.)
4.2.7. Stksize

The Stksize argument must specify the new stack size in bytes:

\[
ok = \text{stksize}(\text{req\_size});
\]

If the return value is OK, then the request for a change of stack size succeeded. If the return value is E\_BADSTKSIZE, then the user has specified an illegal stack size. There are three possible reasons for this error:

1) The requested stack size is too small. The minimum allowed size is 64 bytes. This quantity is known in UNIX parlance as a click, and is the minimum segment size in the 11/60.

2) The requested stack size is too large. The maximum allowed size is 8K bytes (128 clicks). This quantity is the maximum segment size in the 11/60 virtual memory system.

3) The requested stack size is smaller than the current user process active stack, as indicated by the user stack pointer.

If the requested stack size is larger than the amount of memory available, then the value E\_NOMEMLEFT is returned.

4.2.8. Setpri

Setpri takes as an argument an integer from 0 to NPRI-1, where NPRI is the number of user process priority levels allowed:

\[
ok = \text{setpri}(\text{new\_pri});
\]
The only return values are OK or E_BADPRI, which flags an illegal priority specification.

4.3. Memory management and process switching

Our implementation keeps all code, data, and stack areas resident in memory at one time, for both kernel and user. The 11/60 memory management system provides two modes of operation, kernel and user, and provides a separate virtual memory for each mode. The rendezvous kernel runs in kernel mode, and processes using the kernel primitives (which we call user processes) run in user mode.

4.3.1. Physical memory organization

The physical memory is divided as shown in Figure 4.1. The loader sets the value of _end. The kernel allocation area, between _end and 32K, is where the system data structures are located. The kernel will go into panic mode if the rendezvous data structures cannot fit between _end and 32K. Panic mode causes the kernel to print an appropriate error message on the operator's console, and then busy loop forever.

```
+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+
|                 | kernel          |                 |                 |                 |
| kernel & user   | 6               | kernel          | kernel          | user stacks     | I/O             |
| code            | user alloc.     | kernel stack    |                 |                 | page            |
| data            |                 |                 |                 |                 |                 |
+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+
0                  +_end             | 32K              | 40K              | 248K            | 256K            |
physical address
```

**Figure 4.1.** Physical Memory Organization

The amount of physical memory between 40K and the 248K Unibus address
varies from system to system, so the kernel determines the size of the user stack area as well as clearing it during system startup. The sizeandclear() procedure uses probe() to determine whether or not a given click address exists in physical memory. Probe() attempts to access the address in question, trapping a bus error if the physical memory location does not exist.

The user data area should be distinct from the kernel data area, but this has not been implemented yet.

4.1.2. Virtual memory organization

The PDP 11/60 provides a 64K virtual memory space for both user and kernel modes. The virtual memory is divided into 8 segments, each having a maximum length of 8K bytes and a minimum length of 64 bytes (one click). The user virtual address space is organized as shown in Figure 4.2. Again, in a future implementation the user PDB's will reflect a strict separation the user and kernel data areas.

<table>
<thead>
<tr>
<th>Segment #</th>
<th>Physical Base</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>code</td>
</tr>
<tr>
<td>1</td>
<td>8K</td>
<td>&amp; data</td>
</tr>
<tr>
<td>2</td>
<td>16K</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>24K</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>(unused)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>(unused)</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>stack</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>248K</td>
<td>I/O page</td>
</tr>
</tbody>
</table>

**Figure 4.2. User Virtual Address Map**

The kernel virtual memory organization includes two segments, the use
of which is peculiar to the rendezvous kernel (Fig. 4.3). The kernel changes the spare segment base dynamically as its memory access requirements change. For instance, the kernel returns the result of a rendezvous in the top word of the user stack page. Since a given user stack may be located anywhere in physical memory between 40K and 248K, the kernel sets the PAR for its spare segment so that a write into the a top word of the spare segment maps into the top word of the user stack.

<table>
<thead>
<tr>
<th>Segment #</th>
<th>Physical Base</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>code</td>
</tr>
<tr>
<td>1</td>
<td>8K</td>
<td>data</td>
</tr>
<tr>
<td>2</td>
<td>16K</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>24K</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>32K</td>
<td>stack</td>
</tr>
<tr>
<td>5</td>
<td>(variable)</td>
<td>stack exten</td>
</tr>
<tr>
<td>6</td>
<td>(variable)</td>
<td>spare</td>
</tr>
<tr>
<td>7</td>
<td>248K</td>
<td>I/O page</td>
</tr>
</tbody>
</table>

*Figure 4.3.* Kernel Virtual Address Map

The kernel allocates a *stack extension area* in physical memory for every user process. The stack extension area contains all of the information necessary to restore the state of a user process after an interrupt or system call has taken place. In particular, the kernel stack extension contains user registers r0-r5, the user sp, and the user FS and PC at the time of the call. The stack extension also contains various kernel return addresses and a trapcode. The function of the kernel stack is explained in the section on process switching, below.

4.1.1. System calls and process switching

The kernel stack extension is so named because it extends the top of
the kernel stack into the next segment. The kernel stack segment and the stack extension segment are adjacent; if the kernel sp is pointing at the base of the stack extension segment, a push statement, \texttt{mov X, -(sp)}\texttt{,} brings the sp into the kernel stack proper. The kernel uses this fact to effect fast process switching.

The sequence of actions that take place on a system call are given below. The C files mentioned (low.c, etc.) are listed in Appendix B.

1) User process A invokes one of the rendezvous primitives, say, \texttt{Setpri}.

2) The assembly language routine \_setpri in libc puts the constant SETPRICODE in r0, and executes a trap instruction.

3) The kernel sp is pointing at the top of the kernel stack extension. That is, the virtual address in the kernel sp is the virtual base address of the stack extension segment plus the size of the kernel stack extension area. The trap instruction causes the user PS and PC to be pushed onto the kernel stack extension.

4) The processor loads its new PS and PC from the syscall (trap instruction) vector location. The processor is now in kernel mode. Execution resumes at the assembly language syscall handler routine in low.c.

5) The syscall handler pushes registers r0--r5, the user stack pointer, the trapcode, and two subroutine return pointers into the kernel stack extension. (The trapcode is derived from the new PS. It is used by the trap and interrupt handling routines, but is ignored by syscall.) When the syscall handler is finished, the information necessary to
restore the user process lies exactly in the stack extension area, and
the kernel sp points to the virtual base of the stack extension se-
gment. Thus, the next push operation will push data onto the kernel
stack proper address space, rather than into the stack extension
address space. Figure 4.4 shows the situation at this point.

| proc. A stack | in physical
| extension area | memory

^ memory ^

| map |

-----------------------------------------------
kernel || process A ... kernel virtual
_stack || state address space

| ^

| kernel sp ->

**Figure 4.4. After a Call from Process A**

6) The trap handler invokes the C procedure syscall(). syscall() will
perform various validity checks (e.g., examine the user stack pointer
to insure that it points into the user stack segment), and set the
kernel spare segment PAR so that the segment maps to the user's stack
area in physical memory. This means that both the user and kernel
virtual addresses in the nth segment map to the same physical location
in memory. syscall() will then examine r0 to see which primitive was
invoked by the user. Since r0 contains SETPRICODE, syscall will
invoke setpri().

7) setpri() gets its argument from the user stack. The user stack is
available to the kernel primitives because the user stack pointer can
be retrieved from the kernel stack extension area, and at all times the kernel spare segment maps into the physical user stack.

8) setpri() does the appropriate things and then writes its return value (OK or an error code) into the location in the stack extension area where the user's r0 was saved.

9) Since the setpri() call may change the running process's priority, setpri() invokes the scheduler in low.c. The scheduler selects a user process B to run, and resets the kernel stack extension PAR so that the stack extension segment is positioned over the user process B's stack extension area in memory. In addition, the user stack PDK and PAR registers are set to reflect the physical location and length of user B's stack.

10) The dispatcher is now invoked. Since every kernel routine cleans up its stack, and all kernel routines are now completed, the kernel up again points to the bottom of the stack extension (which maps into the B's physical stack extension area). The memory situation is shown in Figure 4.5.

The dispatcher pops the stack extension values back into r0-r5, the user stack pointer into the user sp, and executes an rtt (return from trap) instruction. The rtt will load the user PS and PC from values popped off the kernel stack.

11) The process switch has now been effected, and user B runs.

Aside from a few Runlist[] manipulations, which we discuss in the section on data structures, the entire process switching overhead consists of
changing the kernel stack extension PAR and the user stack PAR and PDR.
The user process state save and restoration must take place on every system
call and interrupt, even if the calling user process continues to run.

4.3.4. **Interrupts, traps, and more on process switching**

If an interrupt takes place while a user process is running, the
interrupt has the same stack effect as an rtt instruction: the user FS and
PC are pushed onto the kernel stack. The new FS and new PC are loaded from
the device interrupt vector address, and the PC points to the assembly
language interrupt handler routine in low.c, where execution resumes.

The interrupt handler routine proceeds exactly as does the syscall
handler routine, pushing the user state on the stack extension. Instead of
invoking syscall(), however, the interrupt handler invokes the C routine
interruptv(). After interruptv() has done all of the appropriate things,
the scheduler is invoked, the dispatcher is invoked, and we return to the
user process with an rtt.

One might expect, perhaps, that we would return to the user process from an interrupt with an rti (return from interrupt) instruction. However, the assembly routines in low.c that handle system calls, interrupts, and traps all share the same code except for a few entry instructions. One could write handlers for each event, but we did not feel compelled to do so. The penalty for using rtt instead of rti, for instance, is the loss of a trap trace facility. This did not concern us greatly. In addition, interruptrv() and trap() do use the trapcode, and syscall() does not. This results in some useless information being pushed onto the kernel stack extension for system calls. It remains to be seen whether or not the performance of the system could be enhanced by the removal of the extraneous push and pop on a system call.

If an interrupt takes place while the processor is in kernel mode, then no attempt is made to switch the user process. After interruptrv() runs, the dispatcher merely executes the return sequence, restoring the processor to the state it had when the interrupt took place.

If a trap occurs, the same state push sequence occurs as with system calls and interrupts. However, the C routine trap() is called instead of syscall() or interruptrv(). As it stands, trap() does not deal with traps very gracefully; it merely prints the trapcode and other relevant information on the operator's console, and busy loops forever.

4.4. Data Structures

There are five main data structures used in the kernel: Process Entries, Runlist, the Association Table, Rendezvous Classes, and Rendezvous
Table Entries. Each structure is described in detail below. All of these data structures are declared in the header file proc.h (see Appendix B).

Most of the structures involve list linking in one way or another. listhdr is a very common substructure, being defined in C as shown in Figure 4.6. The flink points to the the structure ahead of this one on the linked list, while blink points to the structure behind this one. If a listhdr is in node X, and the flink and blink point to node X, then X is the only node on the list.

```
struct listhdr {
    struct listhdr * flink;
    struct listhdr * blink;
};
```

Figure 4.6. List Header Structure

Two assembly language routines operate on structures that have listhdr as their first field. Addtolist( np,lp ) inserts the node pointed to by np onto the list pointed to by lp. Rmfromlist( np ) removes the node pointed to by np from the list that it's on.

4.4.1. Runlist

The Runlist is an array of listhdrs. Runlist[i] is the head of a list of Process Entries (PE's) that are currently runnable at priority i. One Process Entry node exists for every user process. We have chosen to implement run-to-completion, highest-priority-first processing. The current process runs until it either exits or blocks on a rendezvous. At that time the scheduler selects the highest priority process from the Process Entries
linked on Runlist, removes the PE from Runlist, does the necessary things to effect a process switch, and invokes the dispatcher. The fact that a PE is linked somewhere on Runlist indicates that the process is runnable; blocked processes and the currently running process are not on any list.

4.4.2. Process Entries

When a user process is cloned with fork(), a new Process Entry is created for the clone. A Process Entry contains the following items:

-- **pe_links**, a listhdr allowing the PE to be linked on lists.

-- **pe_pri**, a pointer to Runlist[i] if this process's priority is i.

-- **pe_parusrk**, the PAR for this user process's stack segment.

-- **pe_pdrusrk**, the PDR for this user process's stack segment.

-- **pe_parkatrk**, the PAR for the kernel stack extension segment for this user process.

-- **pe_rvlist[NPRVC]**, a list of pointers to rv_classes for which this process has capabilities.

-- **pe_state**, a code for the state of the user process.

The process state code is PES_RUN for a runnable process, and PES_BLK for a process blocked on a rendezvous request. If the PE node is not allocated to any process, then the code is PES_INACTIVE. The kernel keeps a freelist of inactive PE nodes.

If the process has a capability j to a rv_class, then pe_rvlist[j] points to the Rendezvous Class node for that rv_class. (This is why users
see capabilities as small integers.) pe_rvlist[k] contains NONE if k does
not denote a capability for this process.

When the freelist of PE nodes is created at system startup time, a
one-click area of physical memory is allocated to each PE to serve as a
kernel stack extension area for the PE process. The pe_parkstkx field con-
tains the kernel mode PAR to be used for the stack extension segment when
that process is running. The pe_parkstkx field is a permanent fixture in
that PE; even when a PE is put back on the freelist, its stack extension
area remains allocated, and its pe_parkstkx field remains untouched.

4.4.1. Rendezvous Classes

Although rv_classes are created in pairs, there exists one Rendezvous
Class (RVC) node for every rv_class. A RVC contains the following items:

-- **rvc_links**, a listhdr allowing the RVC to be linked on lists.

-- **rvc_flags**, 16 bit flags denoting any special aspects of the RVC.

-- **rvc_magtype**, the type of the messages that can be passed with this
rv_class.

-- **rvc_cnta**, a union type containing either:

  -- **rvc_refcnt**, a reference count of the number of capabilities to
     this rv_class extant among all user processes

  or

  -- **rvc_intcnt**, a count of the number of interrupts that have
     occurred on this rv_class since the last rendezvous on the match-
ing class.

-- rvc_matchp, a pointer to the RVC that matches this RVC.

The rvc_flags field currently contains only one flag. If (rvc_flags and RVCF_DEVICE) is true, then the rv_class for this RVC has been associated with a device by an Assoc call. In that case, rvc_intent of the rvc_cnts field is the selected disjunct. The use of this field will become clearer when we discuss interruptv(), below.

The kernel keeps a freelist of inactive RVC nodes.

4.4.4. Rendezvous Table Entries

If a rendezvous request does not find an immediate match, then the kernel creates a Rendezvous Table Entry (RVE) for every triple in the request having a true condition field. A RVE has the following fields:

-- rve_links, a listhdr allowing the RVE to be linked on lists.

-- rve_clnk, a pointer to a RVE. This field allows the RVE to be linked on a singly-linked list of RVE's.

-- rve_proc, a pointer to the PE of the calling process.

-- rve_msg, the message field of the argument triple.

-- rve_triplen, the triple number of the argument triple.

The rve_links are used to link the RVE onto the list of RVE's waiting for a rendezvous on a RVC. The rve_clnk field is used to link all of the RVE's from a single call together. This two-way linkage is discussed in more detail in the description of the rendezvous(), below.
The kernel keeps a freelist of inactive RVE nodes.

4.4.5. Association Table

The Association Table (AT) is an array relating rv_classes to devices. Asscoctab[j] is a structure containing the following fields:

--- ate_rvc, a pointer to the RVC for assoc code j.

--- ate_vec, a pointer to an interrupt vector. An interrupt through that vector is turned into a rendezvous on the class pointed to by ate_rvc.

By 11/60 paradigms, an interrupt vector contains the PS and PC for the interrupt handler routine. However, if a user has done an Assoc on a particular interrupt vector, then that vector contains an integer j in the low-order 5 bits of its PS that indexes into the Asscoctab. Hence, Asscoctab[j].ate_rvc denotes the rv_class on which the rendezvous should take place.

4.5. How the primitives work

This section gives a brief overview of the operation of each of the C rendezvous primitive routines. Although it is not strictly a primitive, the interruptrv() routine is discussed as well, since it performs functions related to the assoc() and rendezvous() primitives.

4.5.1. create()

The create() routine gets the requested rv_class types from the user stack and checks them for validity. Next, for two empty slots in PE.pe_rvlist[] are sought. That is, i and j are sought such that PE.pe_rvlist[i] = PE.pe_rvlist[j] = NONE. If two new slots cannot be
found, then an error message is returned to the caller in his r0 location on the kernel stack extension. Otherwise, two fresh RVC's are pulled off the freelist. The RVC fields are initialized to the appropriate values (message type, reference count = 1, etc.), and the i and j slots of PF.pe_rvlist[] are set to point to the new RVC's.

The return values i and j are packed into rl, and create() returns. Even though the user calls Create with &cl and &c2, indicating the address where the return values should be placed, the create() primitive cannot easily access the user data area. Instead, the instructions immediately following the trap instruction in lib.c, which invoked the kernel call in the first place, unpack rl and store the new capabilities in cl and c2. This is easily done, since when we return from the trap instruction we are back in user mode. Create is the only primitive that operates this way; all the other primitives return their value to the user in the conventional C manner, in r0.

4.1.2. release()

release() gets the argument capability off the user stack and checks it for validity. If the capability j is owned by the caller, then PE.pe_rvlist[j] points to a RVC. The RVC reference count field of the RVC is decremented. If the reference counts for the RVC and its matching RVC sum to 0, then both RVC's are returned to the freelist. (If, in addition, the matching RVC is associated with a device, then the device interrupt vector is cleared, and the matching RVC is removed from Asscoctab[].) The sum of the new reference counts for RVC and its match are returned to the user in r0.
4.5.3. **atksize()**

`atksize()` gets the requested new size of the user stack off the stack and checks it for validity. The requested size must be at most 8K bytes and at least as large as the current stack, as indicated by the position of the user stack pointer. If the requested stack size is legitimate, then the byte size is converted to clicks. If the requested size is smaller than the current size, then the released space is given back to the memory manager with a call to dealloc() (described below). The `PE.pe_pdrustk` field for the calling process is updated with the new stack PDR. (The PAR remains unchanged.)

If the requested size is larger than the current size, then a block of memory of the desired length is requested from the memory manager with a call to `alloc()` (described below). If enough free memory exists, then `alloc()` will return a location where the new stack area may be based. The existing user stack is copied into the new stack area, and the old stack area is released with a call to `dealloc()`. The `PE.pe_pdrustk` and `PE.pe_parustk` fields for the calling process are updated with the new stack segment PDR and PAR values.

If the `atksize()` request succeeds, then the scheduler is invoked to insure that the new user stack segment PAR and PDR values are actually loaded into the hardware registers from the PE.

4.5.4. **setpri()**

`setpri()` gets the requested priority off the user stack, and checks it for validity. If the requested priority is `j`, then the caller `PE.pe_pri` field is set to point to Runlist[j]. The caller PE is added to the
Runlist[] linked list as a runnable process, and the scheduler is invoked to select the next user process to run.

4.2.5. fork()

fork() determines the size of the caller process's stack by examining the stack PDR, and requests a new block of memory of equal length from the memory manager with a call to alloc(). If there is enough memory, then a child PE node is pulled off the freelist, and its PE_pe_parustk field is set to reflect the location of the child's stack in memory. The parent's stack is copied to the child stack area, and the parent stack extension area is copied to the child stack extension area. However, the child's r0 location in the stack extension is set to CHILD, and parent's r0 is set to PARENT. The remainder of the parent PE is copied into the child PE, and the child PE is put on Runlist[] at the same priority as its parent.

4.2.6. exit()

All of the caller's rendezvous capabilities are released, and the appropriate reference counts decremented, in a manner analogous to that in release(). The user stack area is returned to the memory manager with dealloc(), and the caller PE node is put back on the freelist. The scheduler is invoked.

4.2.7. assoc()

assoc() will get the requested vector address off the user stack and check it for validity. The vector location itself is examined to insure that no other Assoc() has been performed on that location before. If the vector is clear, then an empty slot in Assoctab[] is sought. If
Associab[j].ste_rvc = NONE, then slot j is free. The address of the interrupt handler in low.c is put in the PC vector location, and the FS is packed with j in its low-order 5 bits.

Two new RVC nodes are gotten off the freelist. A capability for one of the RVC nodes is given to the caller; a rendezvous on that capability will correspond to a wait-for-interrupt. The other RVC has its RVC.rvc_flags field set to indicate that this RVC is associated with a device. The RVC's are initialized, and the type fields of both RVC's are set to void. Associab[j].ste_rvc is set to point to the device RVC, and Associab[j].ste_vec is set to point to the vector. The capability j is put in the user's r0.

4.5.8. rendezvous()

The triple count is gotten off the user stack, and checked for validity. For each triple, the triple fields are gotten off the user stack. The condition field is checked; if it is 0, the triple is ignored. Otherwise, the rv_class field is checked for validity. If PE.pe_rvlist[rv_class field] = NONE, then the caller does not own the specified capability, and an error code is returned in r0. Otherwise, the rvlist element points to the specified RVC.

If the matching RVC is of type rv_class, then PE.pe_rvlist[] is checked to insure that the caller has an empty slot to receive a new capability. If no empty slot exists, then an error code is returned. If the triple RVC is of type rv_class, then the triple message field is checked to insure that the caller owns that capability.

If the matching RVC is associated with a device, then the
RVC.rvc_cnts.int_cnt field of the matching RVC is checked. If it is nonzero, then an interrupt from the device has occurred; since this corresponds to a rendezvous on the triple RVC, the interrupt count field of the matching RVC is decremented, and the triple number is returned to the caller in r0.

The rendezvous table does not exist as a distinct entity, but is formed from interleaved RVC and RVE nodes. If a rendezvous request is unsuccessful, a RVE node for each true triple is gotten off the freelist, and linked onto the appropriate RVC. A typical snapshot of the rendezvous table is given in Figure 4.7. In this snapshot, the RVE's A, B, and C are all waiting for a rendezvous on the RVC that matches RVC 1. (Actually, the process denoted by the PE pointed to by RVE(A).rve_procp is waiting for the rendezvous. However, we speak of RVE's waiting for the sake of brevity.) There is no process waiting for a rendezvous on RVC 2. RVE's B, D, and F are linked together, and were created in the same rendezvous call. The process that made the call is currently blocked, and its PE does not appear on Runlist[]. Note that both the horizontal and vertical lists are circularly linked, although we have left this detail out of the diagram for clarity.

The rv_class field of a triple is dereferenced to a RVC. The matching RVC is examined to see if there is a RVE waiting on it. If there is, then we have a rendezvous, and we discuss what happens below. If the matching RVC does not have an RVE waiting on it, then a RVE is pulled off the freelist, and all of the relevant information from the triple is copied into it. The new RVE is linked, using the rve_clink field, to the other RVE's generated so far in this call.
Figure 4.7. Rendezvous Table Snapshot

If all triples have been examined without a match being found, then
the generated RVE nodes must be added to the RVC’s. The RVE nodes are
already linked vertically, and now each RVE is added to the tail of the
horizontal RVE list for the appropriate RVC. The current PE is removed
from Runlist[], and the scheduler is invoked.

If the rendezvous was successful, then all RVE nodes generated so far
are put back on the freelist, as are all RVE’s from the matching process’s
call. If a rv_class was passed by either party in the rendezvous, then a
capability for the appropriate RVC is added to a free slot in PE.pe_rvlist.
(The receiving process is guaranteed to have a free slot, since we do not
put it in the rendezvous table unless it does.) The calling process
receives the successful triple number in rO. The variable returnmsg is
bound to the top word in the user stack segment, and that is where the
caller's return message is placed.

The kernel spare stack segment PAR register is adjusted to map into the matching process's stack area, and the matching process's return message is written into to the top word of his stack area. The kernel stack extension PAR register is adjusted to map into the matching process's stack extension area, and the matching process's triple number is written into the r0 location. Finally, the matching process's PE is inserted back on Runlist[], and the scheduler is invoked if the matching process's priority is greater than the caller's priority.

4.5.2. interrupt()

The interrupt() routine converts interrupts to rendezvous requests. If a user process requests an Assoc on a device, then the interrupt vector PC for that device is set to point to the interrupt handler in low.c, which ultimately calls interrupt(). The user sp, rl, r0, PC, and PS are passed as arguments to interrupt(), along with the trapcode. The trapcode is the initial low-order 5 bits of the vector PS. The 5 bits are stripped off the PS and passed as a separate argument; those bit locations include the condition code for the running process, and are thus extremely volatile.

The trapcode was set by the assoc() routine to index into Assoctab[]. Assoctab[trapcode] points to the RVC associated with the device. If no user process is waiting on the matching RVC, then the interrupt count field of the RVC, RVC.rvc_cnts.int_cnt, is incremented, and the dispatcher is invoked.

If there is a process waiting on the matching RVC, then the waiting RVE and all other RVE's from the same user call are put back on the
freelist. The waiting process is given its triple number by temporarily placing the kernel stack extension PAR over the waiting process's stack extension area and writing into the r0 location. The waiting process receives no return message, since rv_classes acquired with Assoc calls are automatically of type void.

The waiting process is added to the appropriate Runlist[], and the scheduler is invoked if the awakened process has a higher priority than the user process that was running when the interrupt occurred.

4.6. Memory management routines

Our memory management algorithm is the standard dynamic storage allocation scheme. That is, available blocks of memory are kept on a linked list, with the links and length fields appearing at the beginning of a block. The sizeandclear() procedure clears all of physical memory starting at 40K, and crafts it into the first and only node on the memory freelist.

Calls to alloc() specify a block size in clicks. The freelist is searched, using a first fit algorithm, until a node of sufficient size if found. The node size is decreased by the amount of the requested block size, and the location of the new block is returned to the caller.

Calls to dealloc() specify a physical click address and a size in clicks. The released block is crafted into a node on the freelist, and, if possible, the new node is coalesced into existing nodes lying to its right and left in memory.

4.7. Suggestions for improvement

Our Rendezvous kernel implementation is less than ideal at present.
due to our time limitations. Future implementations will, we hope, include
the features that we deliberately left out in order to hasten the kernel
development, and ideas that occurred to us only after the kernel was com-
plete. Our suggestions for future improvements include:

--- Separate of user and kernel data. The kernel data area should be
unaddressable by user processes. The current implementation allows
user processes full access to the kernel's data area. The Fork primi-
tive does not provide a new copy the user data area for the new pro-
cess.

--- Handle traps from user mode. Traps (as opposed to syscalls) from user
mode ought not to cause the system to print a message on the
operator's console and loop forever. A reasonable response might be
to kill the user process that caused the trap. (Some users might not
consider this reasonable, but let the user beware.)

--- Limit the user's view of the I/O page. Currently every user has full
access to the entire I/O page. An Assoc call should give the user
capabilities (in the general sense, not the rv_class sense) for only
certain addresses in the I/O page. The emt instruction might be used
for system I/O requests.

--- Compact memory. Vigorous use of the alloc() and dealloc() routines
could leave small, unallocable holes in memory. Fragmentation could
be ameliorated by compacting memory every now and then. Allocated
blocks are condensed into one end of memory, leaving the other end one
free block.
--- Speed up Rendezvous. Keeping the index of an available slot in a process's capability table in the process's PE would speed up Rendezvous. Currently, the rendezvous() procedure searches for a free slot if the calling process could receive a capability as a Rendezvous return message.

--- Return informative error messages. Current error messages, especially those in Rendezvous, do little to indicate exactly where the user went wrong. For instance, if a user is returned E_BADCAPY from Rendezvous, he does not know which triple specified the unowned capability.
Appendix A

Specification for the Rendezvous primitives

1. The meta-language

We use a small, straightforward metalanguage to describe the primitives. Consider this example:

\[
\text{Rendezvous}(\{<\text{cond}, \text{rv\_class}, \text{msg}>\}^+)\]

Lower case letters indicate variable names. **Boldface** letters indicate keywords for the rendezvous primitives. The angle brackets, \(<\), indicate a group of one or more variables. The variable name foo by itself is equivalent to \(<\text{foo}>\), while \(<\text{foo}, \text{foobar}>\) means "the two variables foo and foobar". If \(<\text{foo, foobar}>\) appears as the target of a assignment statement, it does not mean that the rvalue of the assignment is of the **Type** "pair of variables"; it means that the primitive on the right hand side of the assignment returns two values.

Finally, \(\{\}\)^+ means, "one or more repetitions of the entity between the braces".

2. Rendezvous classes and types

A rendezvous class (or rv\_class, for short) is a capability used to exchange messages between processes. We define a capability as the right of access to an object. Possession of a rv\_class capability by a process gives that process the right of access to the Rendezvous primitive, i.e., allows the process to request a rendezvous.

Rv\_classes are created in pairs, and each member of the pair is said to match the other member of the pair. For a message exchange to take place, two processes must request a rendezvous with matching rv\_classes. That is, one process must request a rendezvous with one member of the rv\_class pair, and another process must request a rendezvous with the matching member of the rv\_class pair.

Each rv\_class allows messages of one type to be exchanged. There are currently three types:
data. The message passed is a one-byte datum.

void, meaning that no data are passed; the fact that a successful rendezvous took place in the message.

rv_class is considered a type. Thus, rv_classes may be passed in a message. The receiving process can use that rv_class to make subsequent rendezvous requests. The sending process retains full use of the rv_class as well. There is no notion of revocation of rv_class capabilities; once process A has passed a rv_class to process B, process A has no control over the use or propagation of that rv_class by process B.

3. The Rendezvous primitive

3.1. Syntax

Rendezvous( [ < cond, rv_class, msg > ]* )

Returns: < trip_num, return_msg, ok >

3.2. Semantics

A rendezvous call is a request for a message exchange. A caller specifies a set of rendezvous classes, along with a message for each class. When the kernel receives two matching rv_classes from two separate calls (a rendezvous), then it swaps the corresponding messages between the two calling processes.

The kernel is passed one or more triples and returns three values. Each argument triple contains:

cond, a boolean expression involving any variables available to the calling process.

rv_class, identifying a rendezvous class.

msg, a message to be passed to the other end of the rendezvous connection. The type of msg is determined by the rv_class.

The kernel associates a triple number with each triple. The triple number is simply the position of the triple in the argument list, counting from zero. That is, the leftmost triple is triple number 0, the next is triple number 1, and so forth.

The conditions in all the triples are evaluated at the time of the call in unspecified order. Those triples having conditions that evaluate to false are disregarded. When we refer to triples in the rest of this appendix, unless otherwise noted we are referring to only those triples having conditions that evaluate to true.

The system kernel maintains the rendezvous table, a list of unmatched
rendezvous requests. The kernel examines the triples in left to right order. For each triple, the kernel checks the rendezvous table to see if any other caller has requested a rendezvous with the matching rv_class of the triple. If no match is found for any triple, the triple number, rv_class, and msg from all triples are entered into the rendezvous table, and the calling process blocks.

Now consider the case in which a match is found for the n_th triple. If the calling process is A, then triple n matches triple m in a previous call by process B. The triple number, the rv_class, and the message (msg_B) from triple m are in the rendezvous table. The kernel will:

1) If the type of msg_A is not void, return msg_A to process B (the return_msg value).
2) Wake up process B and return to B (the trip_num value).
3) If the type of msg_B is not void, return msg_B to process A (the return_msg value).
4) Return n to process A (the trip_num value).
5) Remove all of process B's rendezvous requests from the rendezvous table.

If the returned value ok is < 0, then the Rendezvous call was erroneous, and a rendezvous did not take place. Possible errors include: 1) For one or more triples the calling process does not own the specified rv_class.
2) All conditions have evaluated to false at the time of the call.

4. Acquiring and releasing rendezvous classes

4.1. Syntax

Create( typel, type2 )

Returns: < rv_class1, rv_class2, ok >

Release( rv_class )

Returns: < num_left, ok >

4.2. Semantics

The Create primitive will return capabilities for two rendezvous classes with the names rv_class1 and rv_class2. Typel and type2 specify the message types that can be passed with a rendezvous request with rv_class1 and rv_class2, respectively. A nonzero ok value is returned iff the Create request did not succeed. Possible reasons for failure are: 1) Illegal type specification. 2) Not enough room in this process's capability table.
Rv_class1 and rv_class2 are local names. Thus, in order to rendezvous using those rv_classes, the process making the Create request must pass either or both of the new rv_classes to another process. There are two ways to pass a rv_class:

1) Start up a child process (with the Fork primitive, described below). The child inherits all capabilities owned by the parent.

2) Send the rv_class to another process in a rendezvous message.

Release will remove all copies of the calling process's capability for the specified rv_class, preventing future rendezvous requests on that capability by that process. The matching rv_class is not affected. The num_left value will contain the number of copies of the capability for the released rv_class owned by all other processes plus the number of copies of the capability for the matching rv_class extant. If the calling process does not own the specified rv_class, then the ok value is < 0.

5. Spawning a child process

5.1. Syntax

Fork()

Returns: who_am_I

Exit()

5.2. Semantics

The Fork primitive will create a new copy (a child) of the calling process (the parent). The child will execute concurrently with the other system programs. Execution of the child begins at the point immediately after the Fork() call.

The child is an exact copy of the parent (including the parent's memory, environment, capabilities, etc.), with two exceptions:

1) The child has different memory management registers, since the child has its own memory space. The child's memory is an exact copy of the parent's.

2) The value of who_am_I returned to the parent is different from the value returned to the child.

A call to the Exit() primitive kills the calling process; that is, it terminates its execution, releases its capabilities, and removes it from the process table.
6. Treating an interrupt as a rendezvous

6.1. Syntax

\texttt{Assoc( vector address )}

Returns: \texttt{< wfi_rvc, ok >}

6.2. Semantics

The Assoc primitive will create a new rendezvous class pair. One
member of the pair is the rendezvous class \texttt{wfi_rvc}. The other member of
the pair is associated with the specified interrupt vector address. After
the Assoc call is performed, an interrupt at the vector address will gen-
erate a rendezvous that matches \texttt{wfi_rvc}. Hence, in order to wait for an
interrupt from the device corresponding to the vector address, a process
must request a rendezvous on \texttt{wfi_rvc}.

The message type of \texttt{wfi_rvc} is \texttt{void}.

Only one Assoc on a given vector address is allowed in the entire sys-
tem.

The \texttt{ok} return value will be nonzero iff the Assoc call did not succeed
(illegal vector address, no space in the calling process's capability
table, an Assoc on the same vector address has already been performed).

7. Setting a process stack size

7.1. Syntax

\texttt{Stksize( n )}

Returns: \texttt{ok}

7.2. Semantics

The Stksize primitive will grow or shrink the size of the stack area
of the current process to be at least \texttt{n} bytes. The return value \texttt{ok}
flags an error if the requested stack area size would be smaller than the current
stack (as indicated by the process stack pointer).

Stksize copies the current stack if the requested stack area is larger
than the current stack area. The kernel sets the calling process's stack
pointer to reflect the new stack location.

8. Setting a process priority
8.1. Syntax

Setpri( p )

Returns: ok

8.2. Semantics

The Setpri primitive will set the calling process's scheduling priority to p. The identifier ok flags an illegal request.
Appendix B

Rendezvous Kernel Code

The following pages contain the C files that form the Rendezvous kernel. Files are typd on their file name suffix. Header files, containing declarations and macros, are suffixed with "h". Program files are suffixed with "c".
```
# # TS       h/makefile    April 1981
# # Usage: make
# # IDENT= -DCORNELL

COPTS= $[IDENT]
CFLAGS= -c -O $[COPTS]

HFILES= defs.h memmgt.h list.h syscalls.h proc.h stack.h

all: $[HFILES]
defs.h:
touch defs.h
memmgt.h:
touch memmgt.h
list.h:
touch list.h
syscalls.h:
touch syscalls.h
proc.h:
touch proc.h
stack.h:
touch stack.h

print:
print $[HFILES]

depend:
grep -^#include $[HFILES] | sed 's/#include */ */' | sed 's/^/ /' > makedep
echo -^# DO NOT DELETE THIS LINE/+/2,$$d' > eddep
echo -$$r makedep' >> eddep
echo 'w' >> eddep
cp Makefile Makefile.bak
ed Makefile < eddep
rm eddep makedep
echo -^# DEPENDENCIES MUST END AT END OF FILE' >> Makefile
echo -^# IF YOU PUT STUFF HERE IT WILL GO AWAY' >> Makefile
echo -^# see make depend above' >> Makefile

# DO NOT DELETE THIS LINE -- make depend uses it

memmgt.h: ../h/defs.h
```
proc.h: ../../../defs.h
proc.h: ../../../list.h
stack.h: ../../../defs.h

# DEPENDENCIES MUST END AT END OF FILE
# IF YOU PUT STUFF HERE IT WILL GO AWAY
# see make depend above
# makefile

# Usage: make rv
#
# IDENT= -DCORNELL
#
# COPTS= $(IDENT)
# CFLAGS= -c -O $(COPTS)

FILES= ./lows.o access.o addtolist.o assoc.c clrclick.o create.o exit.o \ 
      fork.o interruptrv.o kalloc.o lib.o main.o memmgmt.o \ 
      parpdrops.o peops.o prf.o probe.o \ 
      release.o rendezvous.o rmfromlist.o root.o rv cops.o \ 
      rvealloc.o rv eops.o \ 
      setpri.o stksize.o trap.o

CFILES= . ./access.c addtolist.c assoc.c clrclick.c create.c exit.c \ 
      fork.c interruptrv.c kalloc.c lib.c main.c memmgmt.c \ 
      parpdrops.c peops.c prf.c probe.c \ 
      release.c rendezvous.c rmfromlist.c root.c rv cops.c \ 
      rvealloc.c rv eops.c \ 
      setpri.c stksize.c trap.c

system:
    cd ..; make
    make rv

rv: $(FILES)
    ld -o rv $(FILES)
    size rv
    chmod 755 rv
    ls -1 rv

access.o: access.c cpp_asm.sed
    cc -E access.c | sed -f cpp_asm.sed > access.s
    as -o access.o access.s
    rm access.s

addtolist.o: addtolist.c cpp_asm.sed
    cc -E addtolist.c | sed -f cpp_asm.sed > addtolist.s
    as -o addtolist.o addtolist.s
    rm addtolist.s

lib.o: lib.c cpp_asm.sed
    cc -E lib.c | sed -f cpp_asm.sed > lib.s
    as -o lib.o lib.s
    rm lib.s

lows.o: lows.c cpp_asm.sed
    cc -E lows.c | sed -f cpp_asm.sed > lows.s
    as -o lows.o lows.s
rm low.s

rmfromlist.o: rmfromlist.c cpp_asm.sed
cc -E rmfromlist.c | sed -f cpp_asm.sed > rmfromlist.s
as -o rmfromlist.o rmfromlist.s
rm rmfromlist.s

rvealloc.o: rvealloc.c cpp_asm.sed
cc -E rvealloc.c | sed -f cpp_asm.sed > rvealloc.s
as -o rvealloc.o rvealloc.s
rm rvealloc.s

cpp_asm.sed:
echo '/^\#d' > cpp_asm.sed
echo '/^d' >> cpp_asm.sed

c c $(CFLAGS) $*.c

namelist: rv
nm -gn rv | pr -3 -wl30 -h 'RV Kernel Namelist' | pro

print:
  cd ../h; print *.h
  print $(CFILES)

clean:
  rm *.o

depend:
grep '^[^#]include' $(CFILES) |
  | sed 's/[^(]*\([^]*\)]*$/1/' |
  | sed 's/.c/.o/' >makedep
echo '/^# DO NOT DELETE THIS LINE/+2,$$d' >eddep
echo '$$$r makedep' >>eddep
echo 'w' >>eddep
cp Makefile Makefile.bak
ed Makefile < eddep
rn eddep makedep
echo '# DEPENDENCIES MUST END AT END OF FILE' >> Makefile
echo '# IF YOU PUT STUFF HERE IT WILL GO AWAY' >> Makefile
echo '# see make depend above' >> Makefile

# DO NOT DELETE THIS LINE -- make depend uses it

low.o: ../h/defs.h
low.o: ../h/proc.h
low.o: ../h/list.h
access.o: ../h/defs.h
addtolist.o: ../h/defs.h
addtolist.o: ../h/list.h
assoc.o: ../h/defs.h
assoc.o: ../h/list.h
assoc.o: ../h/proc.h
assoc.o: ../h/stack.h
clrclick.o: ../h/defs.h
create.o: ../h/defs.h
create.o: ../h/proc.h
create.o: ../h/stack.h
exit.o: ../h/defs.h
exit.o: ../h/proc.h
fork.o: ../h/defs.h
fork.o: ../h/proc.h
fork.o: ../h/stack.h
interruptrv.o: ../h/defs.h
interruptrv.o: ../h/proc.h
interruptrv.o: ../h/stack.h
interruptrv.o: ../h/list.h
kalloc.o: ../h/defs.h
lib.o: ../h/defs.h
lib.o: ../h/syscalls.h
main.o: ../h/defs.h
main.o: ../h/proc.h
memmgmt.o: ../h/defs.h
memmgmt.o: ../h/memmgmt.h
parpdrops.o: ../h/defs.h
peops.o: ../h/defs.h
peops.o: ../h/proc.h
peops.o: ../h/list.h
prf.o: ../h/defs.h
probe.o: ../h/defs.h
probe.o: ../h/proc.h
release.o: ../h/defs.h
release.o: ../h/proc.h
release.o: ../h/stack.h
rendezvous.o: ../h/defs.h
rendezvous.o: ../h/proc.h
rendezvous.o: ../h/stack.h
rendezvous.o: ../h/list.h
rfromlist.o: ../h/defs.h
rfromlist.o: ../h/list.h
rvcops.o: ../h/defs.h
rvcops.o: ../h/proc.h
rvcops.o: ../h/list.h
rvealloc.o: ../h/defs.h
rvealloc.o: ../h/proc.h
rvealloc.o: ../h/list.h
rveops.o: ../h/defs.h
rveops.o: ../h/proc.h
rveops.o: ../h/list.h
setpri.o: ../h/defs.h
setpri.o: ../h/proc.h
setpri.o: ../h/stack.h
stksize.o: ../h/defs.h
stksize.o: ../h/proc.h
stksize.o: ../h/stack.h
trap.o: ../h/defs.h
trap.o: ../h/stack.h
trap.o: ../h/proc.h
trap.o: ../h/list.h
trap.o: ../h/syscalls.h
/* DEPENDENCIES MUST END AT END OF FILE*/
/* IF YOU PUT STUFF HERE IT WILL GO AWAY*/
/* see make depend above*/
#ifndef __.DEFS_
#define __DEFS__ yes

/* tunable parameters */
#define STKSIZE 010000 /* size of C stack */

/* fundamental constants -- do not change */
#define NONE 0 /* null pointer */

#endif

/* non-UNIX instructions */

mfp1 = 6500^tst
mtpi = 6600^tst
halt = 0
wait = 1
rtt = 6
reset = 5

/* miscellaneous address constants */

PS = 177776 /* Processor State word */
SW = 177570 /* console SWitch register */

#define PS (*((unsigned *)(0177776))) /* Processor Stateword */
#define SW (*((unsigned *)(0177570)))

typedef unsigned physadr_t; /* physical address type */
typedef unsigned rvmsg_t; /* type of rendezvous message */
typedef char type_t; /* type of rendezvous message type */

#define void int /* void type treated as int for now */

#define BR0 0 /* processor priority 0 */
#define BR1 040 /* processor priority 1 */
#define BR2 0100 /* processor priority 2 */
#define BR3 0140 /* processor priority 3 */
#define BR4 0200 /* processor priority 4 */
#define BR5 0240 /* processor priority 5 */
#define BR6 0300 /* processor priority 6 */
#define BR7 0340 /* processor priority 7 */
#define UPS 0140000          /* User Processor Stateword */
#define IOPAGE 0160000        /* (prev. mode = kernel) */
#define IOPAGEP 076000         /* start addr of i/o page */
#define MMR0 0177572           /* physical start i/o page */
#define MMR2 0177576           /* */
#define UISDRO 0177600         /* mem mgt reg 0 */
#define UISARO 0177640         /* mem mgt reg 2 */
#define KISDRO 0172300         /* user I-space PDR 0 */
#define KISARO 0172340         /* user I-space PAR 0 */
#define KISSRO 0172380         /* kernel I-space PDR 0 */
#define KISSARO 0172400        /* kernel I-space PAR 0 */
#define EXPUP 0                /* EXPand segment UPward */
#define EXPDN 010              /* EXPand segment Downward */
#define NONRES 0               /* */
#define ACCRO 02               /* seg. access NON-RESident */
#define ACCRW 06               /* seg. ACCESS Read Only */
#define SEGLEN 077400          /* seg. ACCESS Read/Write */
#define (max. SEGment LENgth = */
| */ 8K bytes)<<8 */

#define KSTKSEGNO 4            /* Kernel STack SEGment NO. */
#define KSTKVADR 0100000       /* beginning of Kernel STack Virtual */
#define KSTKXSEGNO 5           /* AddrRess space */
#define KSTKXXVADR 0120000     /* */
#define SPSEGNO 6              /* Kernel STack eXtension SEGment NO. */
#define SPSECVADR 0140000      /* beginning of Kernel STack eXtension */
#define VIRTSEGNO 7            /* virtual AddrRess space */
#define ISEGNO 8               /* kernel SPare SEGment NO. */
#define IPOAGEVADR 0160000     /* beginning of SPare SEGment Virtual */
#define ISEGNO 7               /* AddrRess space */
#define USTKSEGNO 6            /* I/O SEGment NO. */
#define USTKVADR 0140000       /* beginning of I/O PAGE Virtual */
#define CLICKSIZE 32           /* */
#define EIGHTK 020000          /* */
#define PSPREVMODE 030000      /* CLICK SIZE in words */
#define MASEUSERMODE 030000    /* */
#define CURRMODE 0140000       /* */
#define ISKERNELMODE 0         /* */
#define NOPENODES 012          /* PS PREVIOUS MODE mask */
#define NORVNCNODES 012        /* */
#define NORVENODES 012         /* previous mode WAS USER MODE value */
#define ASSOCIATION 0          /* ps CURRENT MODE mask */
#define ISKERNELMODE 0         /* current kernel MODE value */
#define NOPENODES 012          /* No. of Process table Entry NODES */
#define NORVNCNODES 012        /* No. of RendezVous Class NODES */
#define NORVENODES 012         /* No. of RendezVous table Entry NODES */
#define NASSOC 040             /* No. of ASSOCIation table entries */
/* GRANULARITY of user proc RESCHEDuling */

/* Various user proc size requirements */
#define NPRI 07 /* No. of user proc PRIority levels */
#define NPRVC 010 /* No. of Private RenderVou## Classes */
#define ROOTSTKSIZ### 010 /* init. ROOT proc STack SIZE, in clicks */

/* User-fault Error messages */
#define E_BADVEC (-1) /* BAD (illegal) VECTor address */
#define E_VECINUSE (-2) /* VECTor has already been associ'ed to */
#define E_BADTYPE (-3) /* BAD TYPE specification */
#define E_BAD CPA# (-4) /* BAD CPAbility specification */
#define E_BADMSGCA## (-5) /* BAD CPAbility specified by the */
/* Message field of the rv triple */
#define E_MPBADMSGCA## (-6) /* attempt to a Proc blocked on the */
/* Matching rv class to send a BAD CPA# */
/* in the MSG field of his triple */
#define E_TRIPCOUNT (-7) /* rv call TRIPLE COUNT is wrong */
#define E_ALLCONDSFALSE (-8) /* ALL triples in the rv call have a */
/* FALSE CONDITION field */
#define E_BADPRI (-9) /* BAD PRIority specification */
#define E_BADSTKSIZ### (-10) /* BAD STack SIZE specification */
#define E_FAULT (-11) /* illegal user stack pointer */
#define E_NOSYS (-12) /* NO such SYSTEM call */

/* Resource Error Messages */
#define E_R_ATAB (-100) /* no room in Association TABLE */
#define E_R_PRVC (-101) /* no room in Private (user) RenderVou## */
/* Class list */
#define E_R_SRV## (-102) /* no System RV Classes left */
#define E_R_NOMEMLEFT (-103) /* NO MEMORY LEFT to fork new process */
/* or grow stacksize */
#define E_R_PENODE (-104) /* no Process Entry NODEs left */
#define E_RMATCHCAPSLOTS (-105) /* no room left in the rv SLO## list for */
/* the proc blocked on the MATCHing */
/* CPA#' */

#define OK 0 /* success value returned by many of */
/* the system calls */

/* Types */
#define T_RV# 01 /* Type is RenderVou## CPAbility */
#define T_DATA 02 /* Type is DATA */
#define T.Void 03 /* Type is VOID */
#define ISLEG##(t) ((t>0) && (t<04))

/* Fork() definitions */
#define PARENT 1 /* the exact values don't matter, as */
#define CHILD

2 /* long as they're different */

#endif

#endif
/**
 * list.h August 1981
/**/

#ifndef _LIST_
define _LIST_ yes

/*
 * Doubly-linked lists.
 * Any structure whose first field is of type struct listhdr
 * is listable.
 */

#ifndef ASM
struct listhdr {
    struct listhdr *flink;
    struct listhdr *blink;
};
define LISTHDRSZ (sizeof(struct listhdr))
#else
define LISTHDRSZ 1hsize
#endif

/* List manipulation */

#ifndef ASM
extern void addtolist( /* nodeptr, listptr */ );
extern struct listhdr *rmfromlist( /* nodeptr */ );
define mkemptylist(lp) (((lp)->flink = (lp)->blink = (lp))
define isempty(lp) (((lp)->flink == (lp)) && ((lp)->blink == (lp)))
#endif

#endif
/*
/* memmgt.h
/*

#ifndef _MEMMGMT_
#define _MEMMGMT_ yes

#ifndef _DEFS_
#include "../h/defs.h"
#endif

/*
 * Structures and definitions for mem mgt routines (alloc and dealloc)
 */

struct header {
    physadr_t next_hdr; /* click address of the next node */
    /* in the freelist */
    unsigned size_hdr; /* size of this node (in clicks) */
};

#define NEXT ((struct header *) SPSEGVADR)->next_hdr
#define SIZE ((struct header *) SPSEGVADR)->size_hdr

#endif
#ifndef _PROC_
define _PROC_ yes

#ifndef _DEFS_
#include "./h/defs.h"
#endif

#ifndef _LIST_
#include "./h/list.h"
#endif

/**
/* Process, Rendezvous Table, and Association Table Structures. */
/**

/*******************************************************************************/
/* Rendezvous Capability structure and operations */

#ifndef ASM
struct rvc {
    struct listhdr rvc_links; /* Header of rcv table element list */
    char rvc_flags; /* Status info -- see below */
    type_t rvc_msgtype; /* Type of message to be passed */
    /* from processes waiting on this */
    /* rvc */

    union {
        int rvc_refcnt; /* Reference count */
        int rvc_intcnt; /* Pending interrupt count for rvc */
        /* associated with device */
    } rvc_cnts;

    struct rvc *rvc_matchp; /* --> matching rvc structure */
};
#endif

/* rvc_flags codes */
#define RVCF_DEVICE 01 /* this rvc associated with a device */
/* rvc_msgtype codes are indefs.h */

/* rvc operations */
#ifndef ASM
extern void rvcinit( /*howmany*/ ); /* Initialize rvc freelist */
struct listhdr rvcreflst;  /* Head of the freelist */
extern struct rvc * rvcbase;  /* base address of rvc array */
extern struct rvc * rvcalloc();  /* Alloc. a rvc node */
extern void rvcfree( /* rvcptr */ );  /* Dealloc. a rvc node */
extern int getrvcpair( /* rvcpl,t1,rvcep2,t2 */ );  /* Get a matched rvc pair */
extern void rvcrlse( /* rvcp */ );  /* Dec. rvc ref count & */
       /* dealloc, if necessary */
#endif

/**************************************************************************/
/* Process Table Entry structure and operations */
**************************************************************************/

#define ASM

struct procentry {  /* Proc proc is listable */
  struct listhdr pe_links;  /* -- > header of run list for this */
  struct listhdr *pe_pri;  /* proc's current priority */
  physadr_t pe_parustk;  /* Far prototype for user stack */
  unsigned pe_pdrustk;  /* Pdr prototype for user stack */
  physadr_t pe_parstkx;  /* Far prototype for kernel stack */
  struct rvc *
    pe_rvlist[NPRVC];  /* Array of rv capabilities */
  unsigned pe_state;  /* Proc entry state (see below) */
};
#define ASM

else

  pesize = 0.
  pe_links = pesize; pesize = pesize + lsize
  pe_pri = pesize; pesize = pesize + 2
  pe_parustk = pesize; pesize = pesize + 2
  pe_pdrustk = pesize; pesize = pesize + 2
  pe_parstkx = pesize; pesize = pesize + 2
  pe_rvlist = pesize; pesize = pesize + [NPRVC*2]
  pe_state = pesize; pesize = pesize + 2
#endif

/* pe state codes */
#define PES_INACTIVE 0 /* PE node not associated with any */
       /* process */
#define PES_RUN 1 /* proc is runnable */
#define PES_BLK 2 /* proc is blocked */

#endif
/* pe operations */

ifdef ASM
extern void pinit( /* howmany */ ); /* Initialize pe freelist */
extern struct procentry * palloc(); /* Alloc. a pe node */
extern void pfree( /* pep */ ); /* Dealloc. a pe node */
endif

/*********************************************/
/* Rendezvous Table Entry structure and operations */
ifdef ASM
struct rventry {
    struct listhdr rve_links; /* Struct rvtable is listable */
    struct rventry *rve_clink; /* Circular link for elts in use */
    /* also used to maintain freelist */

    struct procentry *
        rve_procp; /* --> proc to be awakened */
        rvmsg_t rve_msg; /* Msg to be transmitted */
        int rve_tripleno; /* Triple number of this message */
};

struct rventry *rveavail;
endif

/* rve operations */
ifdef ASM
extern void rveinit( /* howmany */ ); /* Initialize rve freelist */
extern struct rventry * rvealloc(); /* Alloc. a rve node */
extern void rvefree( /* rvep */ ); /* Dealloc. a rve node */
endif

/*********************************************/
/* Association Table Structure */
ifdef ASM
struct trapvector {
    void (* tv_func)(); /* --> Interrupt handler (new pc) */
    unsigned tv_nps; /* New ps. Low order 5 bits are used to */
    /* index into assoctab, below. */
};

struct atentry {
    struct rvc *ate_rvc; /* --> associated rvc structure */
};
struct trapvector *ate_vec; /* --> associated trap vector */
)
)
assoc_table[ASSOC];

#endif

/** Rescheduling countdown */

#ifdef ASM
    int rescheddown;
#endif

#endif
```c
/**
 * stack.h
 /**/

#ifndef STACK_
#define STACK_ yes

#ifndef DEFS_
#include "../h/defs.h"
#endif

/*
 * Kernel stack extension (trapstack) and User stack structures
 */

/**/ Kernel Stack Extension structure */

struct trapstk {
    struct usstk * ts_usp; /* user stack pointer */
    unsigned ts_r1; /* user register 0 */
    unsigned ts_trapcode; /* trapcode (not used by sys calls) */
    unsigned ts_r0; /* user register 1 */
    unsigned ts_opc; /* old pc */
    unsigned ts_ops; /* old ps */
};

struct trapstk *trapstkp;

/**/ User stack structure */

struct usstk {
    int *us_returnpointer; /* Return ptr (not used by sys calls) */
    unsigned us_arg[0]; /* Array of sys call arguments */
};

#endif
```
/*
 * syscalls.h
 */

#ifndef _SYSCALLS_
#define _SYSCALLS_ yes

/* Encodings for the system primitive calls */
#define CREATECODE 0
#define RELEASECODE 01
#define RVCODE 02
#define FORKCODE 03
#define EXITCODE 04
#define ASSOCODE 05
#define SETPRICODE 06
#define STKSIZECODE 07

#endif
#define ASM yes
#include "../h/defs.h"

/*
 * access( padr ), daccess( padr )
 * Set the kernel spare seg. base to point to padr, and return
 * the previous spare seg. par. SPSEGVADR now maps to padr.
 * daccess does not return the previous spare seg par.
 */

_access:
    mov *[$KISARO+[SPSEGNO*2]],r0
    mov 2(sp),*[$KISARO+[SPSEGNO*2]]
    rts pc

_daccess:
    mov 2(sp),*[$KISARO+[SPSEGNO*2]]
    rts pc

*
```c
#include "../h/defs.h"
#include "../h/list.h"

addtolist( np, lp )

struct node *np; /* node of any listable structure type */
/* (node is listable if its first field */
/* is struct listhdr) */
struct list *lp; /* list header */

/ Put np->node on lp->list. lp->list->flink now points to node.
/ Assumes np->node is not on any other list.
/
__addtolist:
mov 2(sp),r0 / np
mov 4(sp),r1 / lp

/ Save r2
mov r2, -(sp)

/ BEGIN critical section
mov *$PS,-(sp)
bis $BR7,*$PS

/ Insert r0->node such that r1->node->flink points to it
mov n_flink(r1),r2
mov r2,n_flink(r0)
mov r0,n_blink(r2)
mov r0,n_flink(r1)
mov r1,n_blink(r0)

/ END critical section
mov (sp)+,*$PS

/ Restore r2
mov (sp)+,r2

rts pc
```
#include "../h/defs.h"
#include "../h/list.h"
#include "../h/proc.h"
#include "../h/stack.h"

/*
 * assoc()
 *
 * Associate an interrupt vector address with a rendezvous class.
 *
 */

assoc()
{
    struct trapvector *vecadr;
    int code;
    int slot;
    struct rvc *rvpl, *rvp2;
    extern tvecлим;
    extern void devinr();

    /* make sure trap vector address is in range */
    /* and not in use */
    vecadr = (struct trapvector *)(trapstkp->ts_usp->us_arg[0]);
    if( (((unsigned)vecadr) & (sizeof(struct trapvector)-1)) != 0 )
        vecadr = (((struct trapvector *)&tvecлим))
        trapstkp->ts_r0 = E_BADVEC;
        return;
    
    if( ((int)(vecadr->tv_func)) != NONE )
        trapstkp->ts_r0 = E_VECINUSE;
        return;

    /* code := index of free slot in assocstab */
    for( code = 0; code < NASSOC; code++ )
        if( assocstab[code].ste_rvc == NONE ) break;

    if( code >= NASSOC )
        trapstkp->ts_r0 = E_R_ATAB;
        return;

    /* slot := index of free slot in current process pe_rvlist */
    slot = getrvslot(currprocp,0);
    if( slot < 0 )
        trapstkp->ts_r0 = E_R_PRVC;
        return;
/* rvp1, rvp2 := two new rvc's */
   if( getrvcpair(&rvp1, T_VOID, &rvp2, T_VOID) < 0 ) {
       trapstk->ts_r0 = E_R_SRVC;
       return;
   }

   /* (at this point resources have been found, and */
   /* we're committed to doing this rather than return */
   /* ing an error) */

   /* set the trap vector */
   vecadr->tv_func = devintr;
   vecadr->tv_nps = BR6+code;

   /* flag *rvp1 as associated with device */
   rvp1->rvc_flags |= RVCF_DEVICE;
   rvp1->rvc_cnts.rvc_intcnt = 0;

   /* set assoctab entry */
   assoctab[code].ate_rvc = rvp1;
   assoctab[code].ate_vec = vecadr;

   /* store rvc in process */
   currproc->pe_rvlist[slot] = rvp2;

   trapstk->ts_r0 = slot;
/**/
/* clrclick.c
/**/

#include "../h/defs.h"

/
/*
 * clrclick( click )
 * Clear the click at the physical click address
 *
*/

void
clrclick(click)
physadr_t click;
{
physadr_t prev;
register int *p;
register int *plim;
physadr_t access();

prev = access(click);
p = ((int *) SPSEGVADR);
plim = ((int *)(SPSEGVADR+CLICKSIZE));
while( p < plim )
   *p++ = 0;

daccess(prev);
}
/*
 * _create.c August 1981
 */

#include "../h/defs.h"
#include "../h/proc.h"
#include "../h/stack.h"

/*
 * _create()
 */

Create a matched pair of rv classes with the specified msg types.

*/

void
_create()
[

   type_t t1, t2;  /* requested msg types for c1 and c2 */
   struct rvc *rvp1, *rvp2;  /* -->actual rv classes allocated */
   int slot1, slot2;  /* slot # of c1 and c2 in user private cap table */

   /* Check requested rvc types */
   t1 = (type_t)(trapstkp->ta_usp->us_arg[1]);
   t2 = (type_t)(trapstkp->ta_usp->us_arg[3]);
   if( !(ISLECTYPE(t1) & ISLECTYPE(t2)) ){
      trapstkp->ts_r0 = E_BADTYPE;
      return;
   }

   /* Find two slots in current process rvlist */
   slot1 = getrvslot(currprocp,0);
   slot2 = getrvslot(currprocp,slot1+1);
   if( slot2 < 0 ){
      trapstkp->ts_r0 = E_R_PRVC;
      return;
   }

   /* Get two new rvc elements */
   if( getrvcpair(&rvp1,t1,&rvp2,t2) < 0 ){
      trapstkp->ts_r0 = E_R_SRVC;
      return;
   }

   /* Put pointers to rvc structures in pe_rvlist */
   currprocp->pe_rvlist[slot1] = rvp1;
   currprocp->pe_rvlist[slot2] = rvp2;

   /* Put local names for new caps in user rl, and return */
   /* OK notice */
trapstkp->ts_x1 = (slot2<<8)+slot1;
trapstkp->ts_x0 = OK;
#include "../h/defs.h"
#include "../h/proc.h"

/*
 * _exit()
 *
 * Kill the calling process.
 *
 */

void _exit()
{
    int slot;  /* slot in process cap'Y list */
    register struct proc *cpp;  /* a copy or currproc in register */

    cpp = currproc;

    /* Release proc capabilities */
    { register struct rvc *rvcp;
      for( slot = 0; slot < NPRVC; slot++ ){
        if( (rvcp = cpp->pe_rvlist[slot]) != NONE ){
          rvclrle( rvcp );
          cpp->pe_rvlist[slot] = NONE;
        }
      }
    }

    /* Deallocate ustk */
    { register int ssize;
      ssize = pdr_sz(cpp->pe_drustk);
      dealloc( parsz_pa(cpp->pe_parustk, ssize), ssize);
    }

    /* Clear pe node */
    cpp->pe_parustk = 0;
    cpp->pe_pdrustk = 0;
    cpp->pe_pri = NONE;
    cpp->pe_state = PES_INACTIVE;

    /* Request rescheduling */
    rescheddown = 0;

    /* Deallocate current pe node (and hence kstk). */
    pefree(cpp);
    currproc = NONE;
}
#include "../h/defs.h"
#include "../h/proc.h"
#include "../h/stack.h"

/*
 * _fork()
 *
 * Create an identical copy of the caller process. Two
 * differences: (1) Child has different upar, kstkxpar.
 * (2) Child has return value CHILD, parent
 * has (oddly enough) return value PARENT.
 */

int _fork()
{
    struct procentry *childpe; /* --› child process entry node */
    int ustksize; /* size of user stack area (in clicks) */
    physadr_t childbase; /* base of child usstk */
    physadr_t access(); /* used to copy parent--›child stack */
    physadr_t savkspare; /* and kstksx */
    int mfpi();
    int *p, *pi;
    int *slot; /* slot in user cap'y table */

    /* Allocate space for new usstk. */
    ustksize = pdr_sz( currproc->pe_pdrustk );
    childbase = alloc(ustksize);
    if( childbase == NONE ){
        trapstk->ts_r0 = E_R_NOMEMLEFT;
        return;
    }

    /* Get new proc entry node, which includes a set kstkxpar */
    /* pointing to an allocated kstkx click. If none avail., */
    /* deallocate usstk. */
    childpe = pealloc();
    if( childpe == NONE ){
        dealloc(childbase,ustksize);
        trapstk->ts_r0 = E_R_PENODE;
        return;
    }

    /* Set CHILD return value and copy ktskx to child. */
    trapstk->ts_r0 = CHILD;
    savkspare = access( childpe->pe_parkstkx );
    p = (int *)SPSEGVAADR;
pi = (int *)KSTMXADR;
while( pi < ((int *) (KSTMXADR+CLICKSIZE*sizeof(int))) ){
    *p++ = *pi++;
}

/* Set up child usstk par,pdr. */
childpe->pe_parustk = spsp_par( usstksize,childbase );
childpe->pe_pdrustk = currprocp->pe_pdrustk;

/* Copy usstk to child. */
daccess(childpe->pe_parustk);
for( p = (int *)(trapstk->ts_uap); p < IOPAGE; p++ ){
    *p = mfpi(p);
}
daccess(savkspare);

/* Copy rest of parent pe --> child pe. Increment ref. */
/* counts of rvc's. */
childpe->pe_state = currprocp->pe_state;
childpe->pe_pri = currprocp->pe_pri;
for( slot = 0; slot < NPRVC; slot++ ){
    childpe->pe_rvlist[slot] = currprocp->pe_rvlist[slot];
    if( currprocp->pe_rvlist[slot] != NONE ){
        currprocp->pe_rvlist[slot]->rvc_cnts.rvcRefCount++;
    }
}

/* Put child on run list (same priority as parent) */
addtolist(childpe, childpe->pe_pri);

/* Return PARENT marker */
trapstk->ts_r0 = PARENT;
/**
 * interruptrv.c
 **/

#include "../h/defs.h"
#include "../h/proc.h"
#include "../h/stack.h"
#include "../h/list.h"

/*
 * interruptrv()
 *
 * Interrupt handler to treat interrupt event as a rendezvous.
 */

interruptrv(usp, rl, code, r0, opc, ops)
int *usp;
unsigned rl;            /* associatab[code] used for this device */
int *code;
unsigned r0;            /* proc
int *opc;
unsigned ops;
{
    unsigned rvcp;            /* -->rvc associated w/ this interrupt */
    struct rvc * wait_rvc;
    struct rventry * wait_rvpe;
    struct procentry * wait_pep;
    struct rventry * oldrvea;

    physadr_t savkstkx;         /* Save kernel stack extension par */

    /* Get PS code and associated rvc class */
    rvcp = associatab[code].pte_rvc;

    /* Panic if no assoc has been done on this class */
    if( rvcp == NONE ) {
        panic("Interruptrv");
    }

    /* If no proc is waiting on the matching rvc, increment */
    /* the interrupt count and return */
    wait_rvcp = rvcp->rvc_matchp;
    if( isempty(wait_rvcp) ){
        rvcp->rvc_cnts.rvc_intcnt++;
        return;
    }

    /* There's a proc waiting. Give it the rventry triple number. */
    wait_rvpe = wait_rvcp->rvc_links.rlink;
    wait_pep = wait_rvpe->rve_proc;p;
    savkstkx = ((int *)KISARO)[KSTKXSEGNO];
(((int *)KISARO)[KSTKXSEGNO] = wait_pep->pe_parkstkx;
trapstk->ts_r0 = wait_rvep->rve_tripleno;
(((int *)KISARO)[KSTKXSEGNO] = savkstkx;

/* Remove the waiting proc's nodes from the rv table */
{ register struct rventry *q;
 q = wait_rvep;
 do{
 (void) rmfromlist(q);
 q = q->rve_clink;
 }while( wait_rvep != q );
}

/* Put the waiting proc's rve nodes back on the freelist */
oldrvea = rveavail;
rveavail = wait_rvep->rve_clink;
wait_rvep->rve_clink = oldrvea;

/* Make waiting proc runnable, possibly reschedule */
wait_pep->pe_state = PES_RUN;
addtolist(wait_pep, wait_pep->pe_pri);
if( wait_pep->pe_pri > currpri ){
    reschedcdown = 0;
}
/ **
/ * kalloc.c
/ **/

#include "../h/defs.h"

/ *
 * kalloc( size )
 *
 * Allocate size bytes of the kernel data area, returning
 * the physical address of the area. Return NONE if no space
 * is available.
 */

extern int end;
static unsigned kmemavail = (unsigned)&end;

int *
kalloc(nbytes)
unsigned nbytes;
{
    int *oldavail;

    oldavail = (int *)kmemavail;
    if( nbytes > ((unsigned)(32*1024)) ) return( NONE );
    kmemavail += nbytes;
    kmemavail += (sizeof(int)-1); kmemavail &= ~(sizeof(int)-1);
    if( kmemavail > ((unsigned)(32*1024)) ) return( NONE );
    return( oldavail );
}
/ * lib.c August 1981 / *

 System call entry library

 #define ASM yes

 #include "../h/defs.h"
 #include "../h/syscalls.h"

 / / create()
 / 
 _create:
    mov $CREATECODE,r0
    sys 0
    clr *2(sp)
    movb rl,*2(sp)
    swab rl
    clr *6(sp)
    movb rl,*6(sp)
    rts pc

 / / release()
 / 
 _release:
    mov $RELEASECODE,r0
    sys 0
    rts pc

 / / rendezvous(), rv() [synonyms]
 / 
 Establish rendezvous return message location
 _returnmsg = IOPAGE-2

 _rendezvous:
 _rv:
    mov $RVCODE,r0
    sys 0
    rts pc

 / / fork()
 / 
 _fork:
    mov $FORKCODE,r0
    sys 0
rts pc
/
/ exit()
/
_exit:
   mov $EXITCODE,r0
   sys 0
/ no return
/
/ assoc()
/
_assoc:
   mov $ASSOC_CODE,r0
   sys 0
   rts pc
/
/ setpri()
/
_setpri:
   mov $SETPRI_CODE,r0
   sys 0
   rts pc
/
/ stksize()
/
_stksize:
   mov $STK_SIZE_CODE,r0
   sys 0
   rts pc
```c
#include "../b/defs.h"
#include "../b/proc.h"
#include "../b/list.h"

jmp *(pc)+
entadr: _entry

/ / trap vectors
/ trap; BR7+0. / (4) bus error
trap; BR7+1. / (10) illegal instruction
trap; BR7+2. / (14) bpt-trace trap
trap; BR7+3. / (20) iot trap
trap; BR7+4. / (24) power fail
trap; BR7+5. / (30) emulator trap
syscall; BR0 / (34) system entry

#if NTM > 0
/ system crash dump -- can be started at either 40 or 44
l: jmp dump
    jmp dump
#endif

/ console -- interrupts go directly to debugging trap handler
/ trap; BR7+7
/ trap; BR7+7
/
/ traps ...
/
trap; BR7+8. / 11/70 parity
trap; BR7+9. / programmed interrupt
trap; BR7+10. / floating point
trap; BR7+11. / segmentation violation
```
_tveclim = .

/ trap vector locations

 Graves: 

// SYSTEM CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:

HERE CODE BEGINS HERE

 Graves:
/ Call C trap handler
  jsr pc,*(r0)+
  KSTKXSZIE = KSTKXSIZE + 2 + CSVSIZE

/ (We're back from the handler now, indirectly
/ through _ihret below)

/ restore USP.
  mtpi sp
/ Restore r1, r0, and return
  mov (sp)+,r1
  tst (sp)+
  mov (sp)+,r0
  rtt

// Set Priority Level from C

_sp10:
  bic  $340,PS
  rts  pc

_sp16:
  bis  $340,PS
  bic  $40,PS
  rts  pc

_sp17:
  bis  $340,PS
  rts  pc

// C procedure call register save //</n

CSVSIZE = 012 /* Saves 4 registers, R2-R5 */

csv:
  mov r5,r0
  mov sp,r5
  mov r4,-(sp)
  mov r3,-(sp)
  mov r2,-(sp)
  jsr pc,(r0)

// SCHEDULER
//
// interrupt and trap handlers "return" by calling here
//
// Note: this code is free to clobber registers r0-r4 and
// not clean up the stack, as it falls thru to cret

/ kernel is entered; reschedule current proc
/ when _reschedcdown <= 0.

/ We want the following to be invariant outside the scheduler
/ critical section:
/
/ ( reschedcdown = 0 ) |
/ ( ( currprocp != NONE ) c\& ( currprocp->pe_pri = currpri ) )
/
/ That is, if it's not time to reschedule, and the caller
/ has not invoked exit, then the caller's priority is the
/ highest priority of any runnable process.
/
/ When the rv routines unblock a proc, and that proc has a higher
/ priority than currprocp (if currprocp exists), then currpri
/ should be set to the priority of the unblocked process, and
/ reschedcdown should be set to 0.
/

__ihret:

/ DIAGNOSTICS
/ .data
/  0: <Dispatching >
/ .text
/  mov $0b,-(sp)
/  jsr pc_{-dbsmsg
/  tst (sp)+
/
/ BEGIN critical section
/  bis $BR6,+$PS
/  IF returning to user mode AND reschedule required
/  bit $PSPREVMODE,+$PS
/  beq 9f
/  dec _reschedcdown
/  bgt 9f
/
/ DIAGNOSTICS
/ .data
/  0: <Rescheduling >
/ .text
/  mov $0b,-(sp)
/  jsr pc_{-dbsmsg
/  tst (sp)+
/
/ THEN invoke the scheduler
/ IF (r2 := currprocp) != NONE
/ mov _currprocp,r2
/ beq 0f
/ THEN replace current process on its run list
  mov pe_pri(r2),r1
  mov fink(r1),r3
  mov r2,fink(r1)
  mov r2,blink(r3)
  mov r3,fink(r2)
  mov r1,blink(r2)

0: / FI

/ r3 := highest priority nonempty runlist
/ archloop: DO
0: / r2 := NPRI
  mov $[NPRI],r2
  r3 := MAXPRI
  mov $[_runlist+[(NPRI-1)*LISTHDRSZ]],r3
1: / DO
  / IF r3 is a nonempty list
  cmp r3,(r3)
  / THEN LEAVE archloop
  bne 2f
  / FI
  / r3 := r3 - 1
  sub $[LISTHDRSZ],r3
  / WHILE( r2-- > 0 )
  sob r2,1b
  / OD
  / wait for next interrupt
  / Note: since we are in this search loop,
  / we are about to return to the user process
  / and drop the CPU priority to 0 anyway, so
  / it should be okay to do it here briefly.
  / It is important that we do it BEFORE
  / either selecting the next process to run
  / or setting _rescheddown.
  bic $BR7,$PS
  wait
  bis $BR6,$PS
  / WHILE true
  br 0b
2: / OD

/ schedule first process on (r3->runlist)
/ r2 := remove first proceentry from r3->runlist
  mov blink(r3),r2
  mov blink(r2),r1
  mov r1,blink(r3)
  mov r3,fink(r1)
  clr blink(r2)
  clr fink(r2)

/ load katzpar, uskpar, uskpdr from r2->proceentry
  mov pe_parkstkx(r2),*$[KISARO+[KSTKSEGNO*2]]
  mov pe_parustk(r2),*$[UISARO+[USTKSEGNO*2]]
  mov pe_pdrustk(r2),*$[UISDR0+[USTKSEGNO*2]]
mov r2, _currproc
mov r3, _currpri
// reset rescheduling countdown
mov $[RESCHED_GRANULARITY], _reschedcdown
9:  // FI
    // (fall through to cret)

///////////////////////////////////////////////////////////////////////////
// C procedure call register restore //
///////////////////////////////////////////////////////////////////////////

cret:
    mov r5, r1
    mov -(r1), r4
    mov -(r1), r3
    mov -(r1), r2
    mov r5, sp
    mov (sp)+, r5
    rts pc

///////////////////////////////////////////////////////////////////////////
// mov from previous space //
///////////////////////////////////////////////////////////////////////////

_mfpi:
    mfpi *(sp)
    mov *(sp)+, r0
    rts pc

///////////////////////////////////////////////////////////////////////////
// halt -- for debug only //
///////////////////////////////////////////////////////////////////////////

_halt:
    // halt
    rts pc

///////////////////////////////////////////////////////////////////////////
// dbmsg -- for debug only //
// preserves all registers //
///////////////////////////////////////////////////////////////////////////

dbmsg:
    mov r0, -(sp)
    mov r1, -(sp)
    mov 6(sp), r0
    mov r0, -(sp)
    jsr pc, _printf
    tst (sp)+
/   halt
mov (sp)+,r1
mov (sp)+,r0
rts pc

// _probeinc -- increment a global location if called  //
// used by memory probe routine for sizedclear       //
///------------------------------------------------------------------
_probeinc:
   inc _probeflag
   rtt

///------------------------------------------------------------------
/ System startup
///------------------------------------------------------------------

_reentry:
/ We get here only on branching to location 0 in kernel mode.
/ Simulate a trap type 10, from kernel mode with return to 0.
   clr -(sp)  
   clr -(sp)  
   mov $BR7+12,*,PS  
   jmp trap

__entry:

/ send init on Unibus (tm)
   reset

/ protect against re-entry
   mov $reentry,entadr

/ set kernel sp to the top of the kernel stack
   mov $[KSTKVADR+EIGHTK],sp

/ clear BSS
   mov $_edata,r0  
0:   clr (r0)+
    cmp r0,sp
    ble 0b

/ Kernel virtual memory:
/                     physical base   segment use
/                     ---------   ---------------
/ Seg. 0          0          -->  
/ 1                8K          |  kernel code & data
/ 2                16K         |          
/ 3                24K         -->
/ 4                _end        kernel stack
/ 5                kernel stack, extension
Set kernel mode PAR and PDR. blocks_left is the length of the
kernel code and data in 64-byte blocks. phys_base is the
physical base addr of the segment.

First clear all PAR,PDR pairs
mov $KISAR0,r1
mov $KISDR0,r2
mov $UISAR0,r3
mov $UISDR0,r4
mov $8,,r0
0:  clr (r1)+
    clr (r2)+
    clr (r3)+
    clr (r4)+
sob r0,0b

Init PAR, PDR pointers; phys_base := 0
mov $KISAR0,r1
mov $KISDR0,r2
clr r3

Set up kernel mode segmentation registers
blocks_left := ( end of kernel rounded up to the
    nearest 64-byte block )/64
(Note the code could be much simpler if it assumed we were
always going to use 32K as the end of the kernel. This code
is fully general ... )
mov $[KSTKSENO]+[EIGHTK],r0
add $63,,r0
als $-6,r0
bic $[101777],r0

WHILE( blocks_left > 0 ) {  
  0:  tst r0
     ble 4f
/ PAR := phys_base
mov r3,(r1)+
/ IF( blocks_left > 0200 )
  cmp r0,$0200
  ble 1f
/ THEN PDR seg. size := 128 blocks; phys_base :=+ 0200
  mov $SEGLEN,(r2)
  add $0200,r3
  br 2f
/ ELSE PDR seg. size := blocks_left; phys_base :=+ blocks_left
1:  movb r0,l(r2) /* put size in high order PDR byte
     add r0,r3
/ FI
/ PDR access := r/w
2:  b 1 $ACCREW,(r2)+
/ blocks_left := 0200
   sub $0200,r0
   br 0b
4:
/}

/ Set PAR, PDR for kernel stack
   mov r3,*$[KISAR0+$[KSTKSEGNO+2]]
   mov $[EXPDS | ACCRW],*[$[KISDR0+$[KSTKSEGNO+2]]]

/ Set PDR for kernel stack extension: 1 click length, r/w
   mov $[ACCREW],*[$[KISDR0+$[KSTKSEGNO+2]]]

/ Set PDR for spare segment: 8K length, r/w
   mov $[SEGLEN | ACCRW],*[$[KISDR0+$[SPSEGNO+2]]]

/ Set PAR, PDR for the I/O page
   mov $07600,*$[KISAR0+$[IOSEGNO+2]]
   mov $[SEGLEN | ACCRW],*[$[KISDR0+$[IOSEGNO+2]]]

/ Turn on mem, mgmt unit
   mov $1,*$HOPO

/ Make user mode the previous mode
   mov $WASUSERMODE,*$FS

/ Call a C routine to size and clear mem, handcraft the root
/ user process and, set ustkpar,ustkldr, and kstkxpar registers.
/ pc,main

/ return is error; _main will call _entry, below
.data
0:  <return from main >
   .text
   mov $0b,-(sp)
   jsr pc, _panic
   halt
/
/ _entry(start_address)
/
/ called by _main to dispatch first user process
/
/ _entry:
/ r2 := user code starting address
   mov 2(sp),r2

/ Set user sp (initially points to the user's rendezvous return
/ msg location)
   mov $[IOPAGE-2],-(sp)
   mtpi sp
/ Set kernel sp to point to top of kstk extension
    mov $[KSTKXSIZE+KSTKXVADR],sp

/ Change to user proc
    mov $UPS,-(sp)
    mov r2,-(sp)
    rtt

/ no return
/* main.c */

#include "h/defs.h"
#include "h/proc.h"

/*
 * main()
 *
 * Handcraft the first user process, create freelists,
 * initialize memory and runlists.
 */

main()
{
extern root();
int i;
physadr_t pcs;

/* Clear the user stack area of physical memory */
sizemclear((5*RIGHT>>6) & 01777);
printf("Memory sized and cleared.");

/* Create empty priority lists, set rescheduling countdown */
for( i = 0; i < NPRI; i++ ){
  mkemptylist( &runlist[i] );
}
reschedcdon = RESCHED_GRANULARITY;
printf("Priority lists created.");

#ifdef UNDEFINED
{ register struct listhdr *p;
  printf("OUNLIST:0");
  for( i = 0; i < NPRI; i++ ){
    p = &runlist[i];
    printf("[%d] at %o, flink=%o, blink=%o, \
       i,p->flink,p->blink");
  }
}
#endif

/* Create the pe, rvc, and rve freelists */
pinit( NOPENODES );

#ifdef UNDEFINED
{ register struct procentry *p, *q;
  printf("PROCENTRIES:0");
  pefree(p = pealloc());
  q = p;
  do {
    printf("%o, flink=%o, blink=%o, parstkx=%o, \
       ");
```c
q=q->flink,q->blink,q->pe_parkstkx);
q = q->flink;
} while( q->flink != p );
#endif

rveinit( NORVENODES );
#endif
ifdef UNDEFINED
{ register struct rventry *p;
p = rveavail;
printf("NOENTRIES:0");
while( p != NONE ) {
 printf(" Zo0,p);
p = p->rve_clink;
}
}
#endif

rvcinit( NORVCNODES );
#endif
ifdef UNDEFINED
{ register struct rvc *p, *q;
printf("NOVC nodes:0");
rvcfree(p=rvcalloc());
q = p;
do {
 printf(" Zo, flink= Zo, blink= Zo, 
q,q->flink,q->blink);
q = q->flink;
} while( q != p );
}
#endif

printf("Freelists created:0");
/* Handcraft the root user proc node */
currproc = pealloc();
if( currproc == NONE ){
 panic("currproc can't be allocated!!");
}
/* root proc runs at lowest priority */
currrpri = currproc->pe_pri = &runlist[0];
/* root proc has 0 capabilities */
for( i = 0; i < NPRVC; i++ ){
currproc->pe_rvlist[i] = NONE;
}
/* root proc is runnable */
currproc->pe_state = PES_RUN;
```
/* Set up root upar, updr */

pcar = alloc( ROOTSTKSIZEx1);
currproc->pe_parusstk = szpe_par( ROOTSTKSIZEx1, pca );
currproc->pe_pdrustk = sz_pdr( ROOTSTKSIZEx1 );

/* Copy user text, data, BSS par, pdr from kernel par, pdr */

{ unsigned parval, pdrval;
  ifdef UNDEFINED
    printf("Segmentation Registers0");
  endif
  for( i = 0 ; i < 8; i++ ) {
    parval = (((int *)KISAROX)[i];
    pdrval = (((int *)KISDRO)[i];
    ((int *)UISAROX)[i] = parval;
    ((int *)UISDRO)[i] = pdrval;

    ifdef UNDEFINED
    printf("par[0], pdr[0]=0, i, parval, i, pdrval");
    endif
  }
}

/* Load upar, updr, kstxxpar registers from root pe node */

((int *)UISAROX)[USTKSEGNO] = currproc->pe_parusstk;
((int *)UISDRO)[USTKSEGNO] = currproc->pe_pdrusstk;
((int *)KISAROX)[KSTXKSEGNO] = currproc->pe_parkstxx;

ifdef UNDEFINED
    printf("ustkpar=0, ustkdr=0, kstkpars=0,  

    ((int *)UISAROX)[USTKSEGNO],((int *)UISDRO)[USTKSEGNO],  

    ((int *)KISAROX)[KSTXSEGNO] ");
endif

ifdef UNDEFINED
{ register unsigned *pi;
  p = (unsigned)currproc;
  printf("Ourproc = 00uourproc:0,p");
  for( i = 0 ; i < sizeof(struct procenter); i++ )
    printf("%s: 00,i,*p++");
}

printf("OS: 00, SP: 00, PS: 00, &csa /- close enough */

endif

/* Enter the user code starting at root() */

printf("User entry at 00, &root");

entry( &root );
/**
 * memmgmt.c
 /**/

#include "../h/defs.h"
#include "../h/memmgmt.h"

static physadr_t base; /* -->base of block freelist */

/**
 * sizeandclear( begin_addr )
 *
 * Clear all of physical memory, starting at the click
 * begin_addr. Set base.SIZE to its size in clicks.
 */

sizeandclear( begin_addr )

physadr_t begin_addr;
{
  unsigned cl_cnt;
  /* click count */
  physadr_t next_cl;
  /* next click to clear */
  int probe();
  int clrclck();

  next_cl = begin_addr;
  cl_cnt = 0;
  /* If the click exists, clear it. Stop at the I/O page. */
  while( (next_cl < IOPAGEP) && probe(next_cl) )
    
  clrclck(next_cl);
  cl_cnt++;
  next_cl++;

  /* Put all physical memory on the first node of the freelist */
  (void) access( base = begin_addr );
  NEXT = begin_addr;
  SIZE = cl_cnt;
}

/**
 * alloc( nlength )
 *
 * Allocate nlength clicks of physical memory. Return the
 * physical click address of the allocated region, or return
 * NONE if there is not enough space.
 */

alloc( nlength )

register unsigned nlength;
{
    unsigned access();
    unsigned savkpar;  /* save spare KPAR */
    extern physadr_t base;
    register physadr_t ch;
    register physadr_t prev;
    physadr_t chnext;
    physadr_t return_val;

    /* node pointer to scan freelist */
    /* prev.NEXT = ch */
    /* temporary holder for ch.NEXT */
    /* return value */

    /* Starting from the first node after base, look for a big */
    /* enough node (or wraparound). (First fit algorithm.) */
    prev = base;
    savkpar = access(base);
    ch = NEXT;
    (void) access(NEXT);
    while( SIZE < nlength && ch != base ) {
        prev = ch;
        ch = NEXT;
        (void) access(NEXT);
    }

    if( SIZE >= nlength ){ /* big enough */
        if( SIZE == nlength ) { /* exact fit */
            if( ch == NEXT )
                return_val = NONE; /* last node, can't allocate */
            else{
                chnext = NEXT;
                (void) access(prev);
                NEXT = base = chnext;
                return_val = ch;
            }
        }else{ /* fits, but not exactly */
            SIZE -= nlength;
            base = ch;
            return_val = ch+SIZE;
        }
    }else /* wraparound; no node big enough */
    return_val = NONE;

    (void) access(savkpar);
    return( return_val );
}
dealloc( nadr, nlength )

Put the nlength clicks beginning at physical address nadr back on the freelist. Coalesce the deallocated memory into an existing adjacent node if possible.

dealloc( nadr, nlength )

physadr_t nadr;
unsigned nlength;
{

unsigned access(); /* base of the freelist */
extern physadr_t base; /* scratch ptr to freelist node */
register physadr_t ch;
physadr_t chnext; /* temporary holders for NEXT and SIZE */
unsigned chsize; /* rvalues (can only access one header */
physadr_t mnext; /* at a time). */
unsigned msize;
unsigned savkpar; /* save spare KPAR */

ch = base;
/* Find the insertion point for the new node. Invariant: */
/* ch > nadr | ch->header,NEXT < nadr & (nadr is not at the */
/* left or right end of memory) */
savkpar = access(ch);
for(;;){
  if( ch < nadr & NEXT > nadr )
    break;
  if( (ch >= NEXT) && ((nadr > ch) || (nadr < NEXT)) )
    break; /* nadr is to the left or right of all nodes */
  (void) access( ch = NEXT);
}

/* The ch node is to the left of nadr, or nadr is at the left */
/* end of memory. Coalesce nadr into the node to the right */
/* of nadr if possible, otherwise form nadr node. */
chnext = NEXT;
chsize = SIZE;
if( nadr+nlength == chnext ) {
  (void) access(chnext);
  chnext = NEXT;
  chsize = SIZE;
  (void) access(nadr);
  SIZE = chsize+nlength;
  NEXT = chnext;
}
else{
  (void) access(nadr);
  NEXT = chnext;
  SIZE = nlength;
base = nadr;

/* Coalesce nadr node into ch node if possible. */
nnext = NEXT;
nsize = SIZE;
(void) access(ch);
if ( ch+SIZE == nadr ) {
    SIZE += nsize;
    NEXT = nnext;
    base = ch;
} else {
    NEXT = nadr;
}

(void) access(savkpar);
/**
 /* parpdrops.c
 /*
 /**/

#include "../h/defs.h"

/*
 * Encode/decode par, and pdr for stack segments (expand down).
 * These could (should?) be macros.
 */

/*
 * sz_pdr( ssize )
 * Return the pdr for a stack segment given the stack size
 * in clicks
 */
unsigned
sz_pdr(ssize)
int ssize;
{  
   return( (unsigned) ((128 - ssize)<<8) | EXPDN | ACCRW );
}

/*
 * pdr_sz( pdr )
 * Return the actual stack size in clicks, given a stack pdr
 */
int
pdr_sz(pdr)
{  
   return( 128 - (int)( pdr>>8 ) );
}

/*
 * szpa_par( ssize, phys adr )
 * Return par for a stack segment with stack size
 * ssize clicks at physical click address phys adr
 */
physadr_t
szpa_par(ssize,phys adr)
int ssize;
physadr_t phys adr;
{  

return( phys adr + (physadr_t)(ssize-128) );
}

/**
 * parsz_pa( par, ssize )
 * Return the physical click address of the stack
 * denoted by stack par with actual stack size ssize
 */
physadr_t
parsz_pa(par,ssize)
physadr_t par;
int ssize;
{
    return( par + (physadr_t)(128-ssize) );
}

/**
 * parpa_sz( par, phys adr )
 * Return the actual stack size of the stack
 * denoted by the stack par and the physical click
 * address of the stack
 */
int
parpa_sz(par,phys adr)
physadr_t par;
physadr_t phys adr;
{
    return( (int)(par) + 128 - (int)(phys adr) );
}


```c
/**
 * peops.c
 /**

#include "..h/defs.h"
#include "..h/list.h"
#include "..h/proc.h"

static struct listhdr pefrelst; /* Head of pe node freelist */

/**
 * getrvslot( pp, min )
 * Return empty rvlist slot number greater than or equal to min
 * in pp->procentry
 */
gtracevslot(pp, min)
register struct procentry *pp;
{
    register int slot;

    for( slot = min; slot < NPRVC; slot++ )
        if( pp->pe_rvlist[slot] == NONE ) return( slot );
    return(-1);
}

/**
 * peinit(howmany)
 * Initialize freelist of pe nodes to hold howmany nodes.
 * Each node includes a permanent kstkxpar value.
 */
void
peinit(howmany)
int howmany;
{
    register struct procentry *
    register physadr_t ppa;
    /* -->physical location of kstkx */

    mkepeemptystkx&pefrelist);
    while( howmany > 0 ) {
        p = (struct procentry *)kalloc(sizeof(struct procentry));
        if( p == NULL ) panic("peinit: kalloc");
        /* Allocate kstkx (size = 1 click) */
        ppa = alloc(1);
        if( ppa == NULL ) panic("peinit: kstkx alloc");
        p->kstkx = ppa;
```
addtolist(p,&pefrelst);
    howmany--;
}
*/

/*
 * pealloc()
 *
 * Allocate a free pe node and return a pointer to it. Return
 * NONE if none are available.
 */
struct procrepry *
pealloc()
{
    register struct procrepry *p;

    if( isempty(&pefrelst) ) return( NONE );
    (void) rmfromlist( p = pefrelst.flink );
    return(p);
}

/*
 * pefree(p)
 *
 * Deallocate the pe node pointed to by p.
 */
void
pefree(p)
struct procrepry *p;
{
    addtolist(p,&pefrelst);
}
```c
#include "./h/defs.h"

/*
 * Address and structure of the
 * KL-11 console device registers.
 */
struct {
  int rsr;
  int rbr;
  int xsr;
  int xbr;
};
#define KL ((int *)&0177560) /* address of console device registers */

#define KLDONE 0200 /* done bit, same in rsr & xsr */
#define KLINKBL 0100 /* interrupt enable, same in rsr & xsr */
#define KLRENBL 0001 /* receiver enable, in rsr */

/*
 * Scaled down version of C Library printf.
 * Only %s %d %x (%d) %o are recognized.
 * Used to print diagnostic information directly on console tty.
 * Since it is not interrupt driven, all system activities are
 * pretty much suspended.
 * Printf should not be used for chit-chat.
 */
printf(fmt, xl)
char fmt[];
{
  register char *s;
  register *adx, c;

  adx = &xl;
loop:
  while((c = *fmt++) != 'Z') {
    if(c == ' ')
      return;
    putchar(c);
  }
  c = *fmt++;
  if(c == 'd' || c == 'l' || c == 'o')
```
printf(*adx, c=='0'? 8: 10);
if(c == 's') {
    s = *adx;
    while(c = *s++)
        putchar(c);
    adx++;
    goto loop;
}

/*
 * Print an unsigned integer in base b.
 */
printf(n,b)
unsigned n,b;
{
    register unsigned a;
    if( (a = n/b) != 0 )
        printf(n,b);
    putchar((n % b) + '0');
}

/*
 * Print a character on console.
 * Attempts to save and restore device
 * status.
 * If the switches are 0177777, all
 * printing is inhibited.
 */
putchar(c)
register c;
{
    if(SW != 0177777) {
        while((KL->xsr & KLDONE) == 0)
            KL->xsr = 0;
        KL->xbr = c;
        if(c == '0')
            while(Kbr48r & KLDONE) == 0)
    }
}

/*
 * getchar()
 * does it by busy waiting on receiver done bit -- whence
 * system is completely hung.
 * echoes, and translates cr to lf.

/

getchar()
{
    register c;

    if( (KL->rsr & KLENBL) == 0 )
        KL->rsr |= KLENBL;
    while( (KL->rsr & KLDONE) == 0 )
        c = (KL->rbr) & 0177;
    if( c == '0';
        putchar( c );
    return( c );
}
/*
 * probe.c
 */

#include "..h/defs.h"
#include "..h/proc.h"

/*
 * probe( click )
 *
 * Test to see if the physical mem. location at click address
 * exists. Will get a bus error trap if it doesn't; bus
 * error trap handler sets probeflag.
 *
 */

int probeflag = 0;  /* Set by trap handler if mem. probe fails */

extern void probeinc();

static struct trapvector buserror, segfault;
static struct trapvector newvec = { probeinc, BR7 };

int probe(click)
unsigned click;
{
unsigned savkpar;

/* establish addressability in spare segment */
savkpar = access(click);
/* set up temporary bus error & seg fault trap handlers */
buserror = *((struct trapvector *) 04);
*((struct trapvector *) 04) = newvec;
segfault = *((struct trapvector *) 0250);
*((struct trapvector *) 0250) = newvec;
/* probe the location */
probeflag = 0;
[ register int y = *((int *)SPSEGVADR); ]
/* restore trap handlers */
*((struct trapvector *) 04) = buserror;
*((struct trapvector *) 0250) = segfault;
/* restore old spare segment */
(void) access(savkpar);

return( probeflag == 0 );
}
/**
/* _release.c
/**/

#include "../h/defs.h"
#include "../h/proc.h"
#include "../h/stack.h"

/

/*
 * _release()
 */

_realse()
{
    register struct rvc *rvp;
    register int slot;

    /* make sure arg is a valid rv capability for the process */
    slot = trapstk->ts_usp->us_arg[0];
    if( (slot < 0) || (slot >= NPRVC) ) {
        trapstk->ts_r0 = E_BADCAPY;
        return;
    }
    rvp = currproc->pe_rvlist[slot];
    if( rvp == NONE ){
        trapstk->ts_r0 = E_BADCAPY;
        return;
    }

    /* return value = # of remaining ref counts to this cap */
    /* and its match */
    trapstk->ts_r0 = (rvp->rvc_cnts.rvc_refcnt)-1
        + rvp->rvc_matchp->rvc_cnts.rvc_refcnt;

    /* take the rvc away from this process */
    currproc->pe_rvlist[slot] = NONE;
    rvclse(rvp);
}

*/
```c
/**
/ * _rendezvous.c
/**

#include "../h/defs.h"
#include "../h/proc.h"
#include "../h/stack.h"
#include "../h/list.h"

/*
 * _rendezvous()
 *
 * The system call Rendezvous.
 */

/* Structure for decomposing the triples on the user stack */

struct stk_triple {
    int t_cond;
    unsigned t_rvslot;
    unsigned t_msg;
};

void
_rendezvous()
{
    struct stk_triple * triple;
    int     tripnum;
    unsigned msg;

    struct rvc * triprrvc;
    struct rvc * matchrvc;
    struct rventry * matchrve;
    struct proentry * matchpe;

    struct rvc * msgrrvc;
    unsigned matchmsg;
    struct rvc * msgmatchrvc;

    struct rventry * q;

    int      totaltrips;
    int      allcondsfalse;
    struct rventry * prve;

    physadr_t savkspare;
    physadr_t savkstxx;

    /* -->Stack triple template */
    /* Number of the current triple */
    /* Message field of tripnum triple */

    /* -->rvc specified by rvslot */
    /* -->rvc matching triprrvc */
    /* -->rventry for matching rvc */
    /* -->proentry in matchrve */

    /* If msg type is RV_CLASS, msgrrvc --> */
    /* rvc specified by msg */
    /* Message field of *matchrvc */
    /* If matchmsg type is RV_CLASS, */
    /* matchmsg-->rvc specified by matchmsg */

    /* Used for adding or removing circular */
    /* rve list from rvc nodes */

    /* Total no. of argument triples */
    /* 0 if at least one triple cond is true*/
    /* -->new rventry node */

    /* Save kernel spare par register */
    /* Save kernel stack extension par */
```
/* Get triple count, and check it for validity */
totaltrips = trapstk->ts_usp->us_arg[0];
if( ((trapstk->ts_usp + sizeof(int))*(2+3*totaltrips)) >= IOPAGE )
   || ( totaltrips <= 0 ) ){
   trapstk->ts_r0 = E_TRIPCOUNT;
   return;
}

/* Set the triple template over the first triple in the stack. */
/* Initialize the rve node circular list pointers. */
   triple = (struct stk_triple *)&(trapstk->ts_usp->us_arg[1]);
   prve = (struct rventry *)NONE;

/* Assume all triples are false until proven otherwise */
   allcondsfalse = 1;

/* Scan all triples, looking for a match in the rv table. */
   for( tripnum = 0; tripnum < totaltrips; tripnum++ ){
      if( triple->t_cond ) {
         allcondsfalse = 0;

         /* Get triple rv_class field, which is actually a slot no. */
         /* in the caller's cap'ya list. Return if caller doesn't */
         /* own the cap'. Get specified rc_class */
         if( triple->t_rvslot >= NPRVC ){
             trapstk->ts_r0 = E_BADCAPY;
             goto out;
         }

         triprvc = currproc->pe_rvlist[triple->t_rvslot];
         if( triprvc == NONE ) {
             trapstk->ts_r0 = E_BADCAPY;
             goto out;
         }

         /* Get matching rvc */
         matchrvc = triprvc->rvc_matchp;

         /* If matching proc is sending a rv_class, make sure */
         /* caller has a rv capability slot open. */
         if( matchrvc->rvc_msgtype == T_RVCAP ){
             if( !ihaveslot() ){
                 trapstk->ts_r0 = E_R_PRVC;
                 goto out;
             }
         }

         /* If caller is sending a rv_class, make sure the msg */
         /* field contains a legitimate capability */
         if( triprvc->rvc_msgtype == T_RVCAP ){
             if( (triple->t_msg >= NPRVC) ||
                 (currproc->pe_rvlist[triple->t_msg] == NONE) ){
                 trapstk->ts_r0 = E_BADCAPY;
             }
if( matchrvc->rvc_flags & RVCF_DEVICE ) {
    /* Matching rvc is associated with a device. */
    /* Succeed if interrupt count > 0 */
    if( matchrvc->rvc_count, rvc_intcnt > 0 )
        goto devmatch;
} else {
    /* Matching rvc not associated with device. */
    /* Succeed if some proc waiting on it. */
    if( !isempty(matchrvc) )
        goto match;
}

/* No proc is blocked on the matching rv_class, so build */
/* an rve node for the triple. Keep a ptr to the triple */
/* rvc in the fink of the rve node for now. */
    prve = rvealloc( prve );
    if( prve == NULL ){
        panic("Rendezvous: No rve nodes left");
    }
    prve->rve_procp = currprocp;
    prve->rve_msg = triple->t_msg;
    prve->rve_tripleno = tripnum;
    (struct rvc *) (prve->rve_links.flink) = triprvc;
} /* fi: "t_cond" */

/* At this point either the condition field is false, or */
/* there is no match for the triple rv_class. Go to the */
/* next triple. */
    triple++;
} /* rof: index tripnum */

/* Return error if all conditions are false */
if( allcondsfalse ) {
    trapstk->ts_r0 = E_ALLCONDSTFALSE;
    goto out;
}

nomatch: ;

/* No match found for any triple. Put the circular list of rve */
/* nodes in the rv table. */
    q = prve;
    do{
        addtolist( q, (struct rvc *)(q->rve_links.flink) );
        q = q->rve_clink;
    } while( q != prve );
/* Take current proc off runlist */
currunproc->pe_state = PES_BLK;
currunproc = NONE;

/* Request rescheduling */
rescheddown = 0;
return;

match: 

/* triprvc, matchrvc --> the matching rv classes */
/* isempty(matchrvc) is false */

/* Get the first entry for the matching rv */
/* class, and the process blocked on that entry. */
matchrve = matchrvc->rvc_links.blink;
matchpe = matchrve->rve_proc;

/* get messages to be passed in both directions */
msg = triple->t_msg;
matchmsg = matchrve->rve_msg;

if( triprvc->rvc_msgtype == T_RVCAP ){
    /* Give the matching proc a cap'y for that */
    /* rv class, and set msg to his cap'y slotno */
    msgrvc = currunproc->pe_rvlist[msg];
    msg = getrvslot( matchpe, 0 );
    matchpe->pe_rvlist[msg] = msgrvc;
    /* Increment rvc's reference count */
    msgrvc->rvc_cnts.rvc_refcnt++;
}

if( matchrvc->rvc_msgtype == T_RVCAP ){
    /* Give caller cap'y for the matchmsg rv_class, */
    /* set matchmsg to its slot no. */
    msgrvc = matchpe->pe_rvlist[matchmsg];
    matchmsg = getrvslot( currunproc, 0 );
    currunproc->pe_rvlist[matchmsg] = msgrvc;
    /* Increment matchrvc's reference count */
    msgrvc->rvc_cnts.rvc_refcnt++;
}

/* Now both msgs are correct. If triprvc is not of */
/* type VOID, give matching proc his msg. */
if( triprvc->rvc_msgtype != T_VOID ){
    savkspare = access( matchpe->pe_parustk );
    ((int *)IOPAGE)[-1] = msg;
    daccess(savkspare);
}

/* Give matching proc his triple no.
 savkstxx = ((int *)KISARO)[KSTKXSEGNO];
 ((int *)KISARO)[KSTKXSEGNO] = matchpe->pe_parkstxx;
 trapstk->ts_r0 = matchrve->rve_tripleno;*/
((int *) KISARO)[KSTKXSEGNO] = savkstxx;

/* Give caller his msg if the type of matchrve is not VOID. */
if( matchrvc->rvc_msgtype != T_VOID )
    ((int *)IOPAGE)[-1] = matchmsg;

/* Give caller his triple number. */
trapstk->ts_r0 = tripnum;

/* Remove match rve nodes from the rv table. */
q = matchrve;
do{
    (void) rmfromlist( q );
    q = q->rve_clink;
} while( q != matchrve );
    { register struct rventry *p;
        p = q->rve_clink;
        q->rve_clink = rveavail;
        rveavail = p;
    }

/* Make matching proc runnable, possibly reschedule */
matchpe->pe_state = PES_RUN;
addtolist( matchpe, matchpe->pe_pri );
if( matchpe->pe_pri > currpri )
    reschedcedown = 0;

goto out;

devmatch: ;

matchrvc->rvc_cnts.rvc_intcnt--; trapstk->ts_r0 = tripnum;

out: ;

if( prve != (struct rventry *)NONE ) {
    { register struct rventry *p;
        p = prve->rve_clink;
        prve->rve_clink = rveavail;
        rveavail = p;
    }
    return;
}

/*
* Test whether current proc has a free slot in its rvlist.
*/
* Later this information will be kept in a procentry field.
 */

int
ihaveslot()
{
    register struct rvc **p, **plim;

    p = & (currprocp->pe_rvlist[0]);
    plim = p + NPRVC;
    while ( p < plim )
        if ( *p++ == (struct rvc *)NONE ) return(1);
    return(0);
}
```c
#define ASM yes
#include "../b/defs.h"
#include "../b/list.h"

/ **
/ rmfromlist.c
/ **/

/ struct node *np; /* --> node, of any listable structure type */
/ / (node is listable if its first field */
/ / * is struct listhdr) */
/ struct list *lp; /* --> list header */
/
/ Remove np->node from np->list, returning the new list head
/(i.e., returns np->node.flink as lp->list)
/
_rmfromlist:

    mov 2(sp),r1 /* np */
/
/* Save r2 */
    mov r2,-(sp)
/
/ BEGIN critical section
    mov *$PS,-(sp)
    bis $BR7,*$PS
/
/* Remove rl->node. Return r0->(rl->node.flink). */
    mov n_blink(r1),r2
    mov n_flink(r1),r0
    mov r0,n_flink(r2)
    mov r2,n_blink(r0)
    cir n_flink(r1)
    cir n_blink(r1)
/
/ END critical section
    mov (ap)+,*$PS
/
/* Restore r2 */
    mov (ap)+,r2
/
/rtsp pc
```
/***/
/ * rvcops.c
/ ***/

#include "../h/defs.h"
#include "../h/proc.h"
#include "../h/list.h"

/*
 * rvcinit(howmany)
 * Initialize freelist of rv classes to hold howmany nodes.
 */
struct rvc *rvcbase = (struct rvc *)NONE;

void
rvcinit(howmany)
int howmany;
{
 register struct rvc *p;
 register int i;

 mkemptylist(&rvcrelist);
 for( i = 0; i < howmany; i++ ) {
   p = (struct rvc *)malloc(sizeof(struct rvc));
   if( p == NONE ) panic("rvcinit");
   if( rvcbase == (struct rvc *)NONE ) rvcbase = p;
   addtolist(p,&rvcrelist);
 }

/*
 * rvcalloc()
 * Allocate a free rvc node and return a pointer to it. Return NONE if
 * none are available.
 */
struct rvc *
rvcalloc()
{
 register struct rvc *p;

 if( isempty(&rvcrelist) ) return( NONE );
 (void) rmfromlist( p = rvcrelist.flink );
 return(p);
}
/* 
* rvcfree(p)
* 
* Deallocate the rvc node pointed to by p.
*/
void
rvcfree(p)
struct rvc *p;
{
    addtolist(p,&rvcfrelst);
}

/*@ 
* getrvcpair( prvpl, tl, prvp2, t2 )
* 
* Allocate and initialize a matched pair of rvc nodes, w/ ref counts = 1.
* The pair have types tl and t2. Set prvpl and prvp2 to point to the pair.
* Return -1 if the rvc nodes can't be allocated, 0 otherwise.
*/
getrvcpair(prvpl,tl,prvp2,t2)
struct rvc **prvpl;
struct rvc **prvp2;
type_t tl, t2;
{
    register struct rvc *rvp1;
    register struct rvc *rvp2;

    /* get pointers to two rvc structures */
    if( (rvp1 = (*prvpl) =rvcalloc()) == NONE ) return(-1);
    if( (rvp2 = (*prvp2) =rvcalloc()) == NONE ) {
        rvcfree(rvp1);
        return(-1);
    }

    /* initialize them */
    mkemptylist(rvp1);
    rvp1->rvc_msgtype = tl;
    rvp1->rvc_flags = 0;
    rvp1->rvc_cnts.rvc_refcnt = 1;
    rvp1->rvc_matchp = rvp2;
    mkemptylist(rvp2);
    rvp2->rvc_msgtype = t2;
    rvp2->rvc_flags = 0;
    rvp2->rvc_cnts.rvc_refcnt = 1;
    rvp2->rvc_matchp = rvp1;

    return(0);
}
void rvcrlse(rvp)
{ register struct rvc * rvp;
   register struct rvc * mrvp;

   /* decrement reference count, return if nonzero */
   if( --(rvp->rvc_cnts.rvc_refcnt) > 0 ) return;

   mrvp = rvp->rvc_matchp;

   /* ref cnts = 0. If matching rvc is assoc. w/ a device, */
   /* free both and return. */
   if( (mrvp->rvc_flags) & RVCF_DEVICE ){
      register struct entry * atep;

      /* find and free assoctab entry, clear trap vector */
      for( atep = &assoctab[0]; atep < &assoctab[NASSOC]; atep++ ) {
         if( atep->ate_rvc == mrvp ) break;
      }
      if( atep == &assoctab[NASSOC] )
         panic("rvcrlse");
      atep->ate_rvc = NONE;
      atep->ate_vec->tv_func = NONE;
      atep->ate_vec->tv_nps = 0;
      atep->ate_vec = NONE;
   } /* free the rvc's */
   rvcfree(mrvp);
   rvcfree(rvp);

   return;

} /* Matching rvc is a reg. rvc. If it also has 0 reference */
/* counts, free both. */
if( mrvp->rvc_cnts.rvc_refcnt == 0 ){
   rvcfree(mrvp);
   rvcfree(rvp);
}
/* rvealloc.c August 1981 */

#define ASM yes

#include "../h/defs.h"
#include "../h/proc.h"
#include "../h/list.h"

/ * p := rvealloc( p ) * /
/ * Given a pointer to a circular rve list, add another node * /
/ * on to the tail of the list and return a pointer to it. * /
/
_rvealloc:

clink = 4

/ * Remove new node from freelist; return NONE if none avail. * /
    mov _rveavail,r0
    beq 9f
    mov clink(r0),_rveavail

/ * IF p = NONE, THEN return single node linked to itself * /
    mov 2(sp),rl; /p
    bne 1f
    mov r0,clink(r0)
    rts pc

/ * ELSE link new node to tail and return with p-->new node * /
1:      mov clink(rl),clink(r0)
    mov r0,clink(rl)
9:      rts pc
/**
/* rveops.c
/**/

#include "../h/defs.h"
#include "../h/proc.h"
#include "../h/list.h"

static struct rventry *rveavail = NONE; /* Head of rve node freelist */

/*
*rveinit( howmany )
*Create a freelist with howmany rv table entry nodes.
*/
void
rveinit(howmany)
int howmany;
{
    register struct rventry *p;

    while( howmany > 0 ) {
        p = (struct rventry *)kalloc(sizeof(struct rventry));
        if( p == NONE ) panic("rveinit");
        p->rve_clink = rveavail; rveavail = p;
        howmany--;
    }
}

/* NOTE: (1) rvealloc() is the assembler file rvealloc.c */
/* (2) Freeing of rve nodes is done explicitly [i.e., there is no rvefree()] by linking the circular list onto the front of the freelist. */
```c
/* _setpri.c */
August 1981
*/

#include "..../h/defs.h"
#include "..../h/proc.h"
#include "..../h/stack.h"

/*
 * _setpri()
 * *
 * Set priority of current process. Request rescheduling.
 * */

_setpri()
{
    register int newpri;
    newpri = trapstkp->ts_usp->us_arg[0];
    if( (newpri >= 0) && (newpri < NPRI) ) {
        currproc->pe_pri = &runlist[newpri];
        trapstkp->ts_r0 = OK;
        rescheddown = 0;
    } else {
        trapstkp->ts_r0 = E_BADPRI;
    }
}
```
```c
#include "../h/defs.h"
#include "../h/proc.h"
#include "../h/stack.h"

/*
* _stksize()
*
* Change the size of the user's stack area to n bytes.
*
*/

void
_stksize()
{
int csize;     /* Current size of user stack (in clicks) */
int rsize;     /* Requested new size (in clicks) */
physadr_t newbase; /* New physical base of user stack */
int * ip;      /* Used to move old-->new stack */
physadr_t savkspare;
physadr_t access();
int mfpi();

/* Get requested size in bytes, check for validity. Must be at */
/* most 8 KB and big enough to hold current usp. */
rsize = (int)(trapstk->ts_usp->us_arg[0]);
printf("rsize(bytes) = %08x,rsize");
if((rsize > EIGHTK) || (rsize < sizeof(int)) ||
   ((unsigned)(IOPAGEVADR-rsize)) >=
   ((unsigned)(trapstk->ts_usp)))
   { trapstk->ts_r0 = E_BADSTKSIZE;
     return;
   }

/* convert requested size to clicks */
rsize += (CLICKSIZE*sizeof(int)-1);
rsize /= (CLICKSIZE*sizeof(int));
printf("rsize(clicks)= %08x,rsize");

/* get current size and compare with requested size */
csize = pdr_sz( currcode->pe_pdrstk );

if( csize == rsize ) /* Stack size unchanged */
   { trapstk->ts_r0 = OK;
     return;
   }
else if( rsize < csize ) /* Stack shrinks */
```
/* Deallocate the released space, and update updr with new size */
dealloc( parsz_pa(currprocp->pe_parustk,csize), csize+raize );
currprocp->pe_pdrustk = sz_pdr(rsize);
}
else{ /* Stack grows */

/* Allocate new stack space if possible */
newbase = alloc(rsize);
if( newbase == NONE ){
    trapstk->ts_r0 = P_R_NOMEMLEFT;
    return;
}

/* Copy old user stack to new stack area */
savkspare = access(newbase+rsize-128);
for( i = trapstk->ts_usp; i < IOPAGE; i++ ){
    *i = mfpi(i);
}
daccess(savkspare);

/* Release old stack area, and update upar, updr */
dealloc( parsz_pa( currprocp->pe_parustk, csize ) );
currprocp->pe_parustk = szpa_par(rsize, newbase);
currprocp->pe_pdrustk = sz_pdr(rsize);
}

/* Insure that updated upar, updr are actually put in hardware */
/* registers by requesting a reschedule. */
reschedcdown = 0;

/* Return OK flag */
trapstk->ts_r0 = OK;
#include "../h/defs.h"
#include "../h/stack.h"
#include "../h/proc.h"
#include "../h/list.h"
#include "../h/syscalls.h"

#define nargs(n) if ( &us->us_arg[n-1]) >= (*((unsigned *)IOPAGEVADR)) goto fs

/*
 * System calls
 *
 * Do some validity checking on arguments,
 * then switch to requested routine.
 */

syscall(usp, rl, code, r0, opc, ops)
struct ustk *usp;
unsigned rl, code, r0, opc, ops;
{
    /* printf("syscall(%o, %o, %o, %o, %o)0, usp, rl, code, r0, opc, ops): 
halt(); */
    
    /* catch system call from kernel mode and panic */
    if ( (ops & CURRMODE) == 1SKERNELMODE )
        panic("System call from kernel");
    /* catch bad user stack pointer (address too low) before */
    /* dereferencing it */
    if ( ((unsigned)usp) < USTKVADR ) goto fault;
    
    /* establish user and kernel stack addressability */
    daaccess( currproc->pe_parustk);
    trapstkp = &usp;
    /* process the system call */
    switch(r0) {
        case CREATECODE:
            nargs(4);
            _create();
            ihret();
            
            case RELEASECODE:
                nargs(1);
                _release();
                ihret();
                
            case RVCODE:
                nargs(1); /* additional checking is done in _rendezvous */
                _rendezvous();
ihret();

    case FORKCODE:
        /* nargs(0); */
        _fork();
        ihret();

    case EXITCODE:
        /* nargs(0); */
        _exit();
        ihret();

    case ASSOCCODE:
        nargs(1);
        _assoc();
        ihret();

    case SETPRICODE:
        nargs(1);
        _setpri();
        ihret();

    case STKSIZECODE:
        nargs(1);
        _stksize();
        ihret();

    default:
        r0 = E_NOSYS;
        ihret();
    }

    fault: ;
    r0 = E_FAULT;
    ihret();

} /*
   * Other traps.
   *
   * We don't yet deal with them very gracefully.
   */

trap(usp, rl, code, r0, opc, ops)
int * usp;
unsigned rl;
int code;
unsigned r0;
int * opc;
unsigned ops;
{   
  printf("FATAL SYSTEM ERROR: trap(%s,%s,%s,%s,%s,%s)0.
         usp,x1,code,r0,opc,ops);  
  for(;); 
}

panic(panicmsg)  
char panicmsg[]; 
{   
  printf("PANIC MODE: %s,panicmsg");  
  for(;); 
}
Bibliography

[ADA 80]


[Brinch Hansen 78]


[Dijkstra 68]


[Hoare 74]


[Hoare 78]


[Ichbiah et al. 79]
