EZ Processes

David R. Hanson and Makoto Kobayashi
Department of Computer Science, Princeton University,
Princeton, New Jersey 08544

Research Report CS-TR-233-89

November 1989

ABSTRACT

EZ is a system that integrates the facilities provided separately by traditional programming languages and operating systems. This integration is accomplished by casting services provided by traditional operating services as *EZ* language features. *EZ* is a high-level string processing language with a persistent memory. Traditional file and directory services are provided by *EZ*'s strings and associative tables, and tables are also used for procedure activations. This paper describes processes in *EZ*, which are procedure activations that execute concurrently and share the same, persistent virtual address space. They are semantically similar to 'threads' or 'lightweight processes' in some operating systems. Processes are values. They are just associative tables and have all of the characteristics of other EZ values, including persistence. Examples of their use and a brief overview of their implementation are included.

This paper will appear in the *Proceedings of the 1990 International Conference on Computer Languages*, New Orleans, March 1990.

EZ Processes

1. Introduction

EZ[1, 2] is a system that integrates the facilities provided separately by traditional programming languages and operating systems. This integration is accomplished by casting services provided by traditional operating systems as *EZ* language features. The result is a system that is intended to be a complete computing environment that—ultimately—can replace both conventional languages and operating systems.

EZ is a high-level string processing language with a persistent memory in which values exist indefinitely or until changed. Traditional file and directory services are provided by *EZ*'s strings and associative tables [1]. Associative tables are also used for procedure activations, providing low-level services such as activation record creation, references to local variables, and access to state information. This approach permits traditionally separate editing and debugging services both to be provided by an editor written in *EZ* that edits *EZ* values [2].

The design challenge in *EZ* is finding the natural 'linguistic encapsulation' of operating system facilities. Earlier work focussed first on encapsulating file system facilities as strings and tables, then on accessing the details of sequential execution by casting procedure activations as tables. This paper describes the encapsulation of processes in *EZ*, which permits *EZ* to provide some of the functions of conventional operating systems at the programming-language level. Sections 2 and 3 briefly describe the *EZ* language and procedures. Processes and I/O are described in §4, the implementation is sketched in §5, and §6 summarizes experience to date and current and future work.

2. The Language

EZ is a high-level string processing language that has its roots in Icon [3] and its predecessors. It has late binding times and considerable run-time flexibility. For example, variables can be assigned values of any type, automatic conversions are performed for most operations, and structures are heterogeneous.

EZ has four basic types of values: numerics, strings, procedures, and tables. Numerics (integers and reals) and strings (of arbitrary length) are scalars, and tables are heterogeneous, one-dimensional associative arrays that can be indexed by and can contain arbitrary values. Procedures are described below.

Since values persist until changed, assigning strings to variables is like creating 'files' in conventional systems. Substring access and assignment provide random access facilities; `s[i:j]` specifies the substring of `s` between character positions `i` and `j`. Tables are like directories in conventional operating systems. For example, the fragment

```
paper["title"] = "EZ Processes"
paper["authors"] = drh || "\n" || mk
```

creates a table for a paper and assigns it to `paper`, and assigns two entries to the new table (double vertical bars, `||`, denote string concatenation). The notation *e.id* is equivalent to *e*`["`*id*`"]` so tables can be used as records, too. Tables can contain tables; for example, if `references` is a table containing entries for a reference list,

```
paper.references = references
```

adds the reference list to `paper`. Tables can be used to construct hierarchical directories, as in UNIX, as well as arbitrary cyclical structures. Except as noted below, tables expand automatically to accommodate their contents. Entries are removed by the built-in procedure `remove`, so `remove(paper,"title")` removes the entry for `title` in the example above.

Expressions usually compute values, but, as in Icon, some expressions may fail to yield values. For example, the relational operators return their right operand only if the relation is satisfied. The absence of values drives control structures such as `if` statements and `for` and `while` loops. For example,

```
sum = 0
for (i = 1; i <= 10; i += 1)
    sum += x[i]
```

computes the sum of `x[1]` through `x[10]`. As long as `i` is less than or equal to `10`, `i <= 10` yields `10` and the loop body is executed. When `i` becomes `11`, `i <= 10` fails to yield a value, which terminates the loop. For some operators, such as assignment, the absence of a value inhibits the execution of the operation. Thus, `max = max < a` updates `max` only if `a` is greater than `max`.

File system operations are usually provided by 'utilities' in conventional operating systems. Such programs often reduce to simple code fragments in *EZ*. For example, the 'list directory' utility in UNIX, `ls`, lists the names in a directory, and the *EZ* equivalent can be done by

```
for (i in paper)
    s = s || " " || i
```

which assigns `" title authors references"` to `s`. The `for–in` loop sequences through the table `paper` assigning the value of each 'index' (which can be of any type) to `i` and executing the loop body.

Automatic conversions between data types obviate the need for 'conversion' functions and utilities found in conventional languages and operating systems. Numeric operators convert their operands to integers or reals as necessary. Similarly, operands of string operators are converted to strings as necessary. For some operators, the operation performed depends on the type of the operands. For example, the relational operators perform lexical comparison if both operands are strings and numeric comparison (with the appropriate conversions) if either operand is numeric.

Conversions between tables and strings are also provided. Tables are converted to strings by concatenating their elements, and strings are converted to tables by constructing a table with the string associated with the index `1`. Thus, for example, simply typing the name of a table displays its contents.

3. Procedures

Procedures are values that contain executable code; a procedure 'declaration' amounts to an assignment of the procedure 'constant' to the identifier. The fragment

```
procedure ls(t) local i, s = ""
    for (i in t)
        s = s || " " || i
    return s
end
```

assigns to `ls` a procedure that returns a list of the indices in the table passed as the argument. Conversions between procedures and strings are performed automatically. Procedures are converted to strings by returning their source code. Thus, entering `"ls"` displays its code. Strings are converted to procedures by compiling them; in a sense, compilation is an optimization in *EZ*. Scope rules depend on tables interrogated by the compiler. Unlike other systems, these 'symbol tables' are *EZ* tables, which can be manipulated at the source-language level. This facility is described in more detail below and in Reference 2.

Procedures can be invoked as usual, for instance,

```
list = ls(paper)
```

assigns `" title authors references"` to `list`.

In addition, conversion from a procedure to a table, provided by the built-in function `table`, yields a table that is an activation record for the procedure. This table contains entries for each of the arguments and locals declared in the procedure and entries describing the current state of the activation.

Arguments and locals in an activation can be accessed by indexing the table. For example, after executing

```
d = table(ls)
```

`d.i` and `d.s` access the local variables `i` and `s` in that activation of `ls`. Such activations may be invoked like procedures, so both

```
d(paper)
```

and

```
d.t = paper
d()
```

begin execution of `ls` with `paper` as its argument.

In addition to the arguments and locals, activations also have the index `Procedure`, which contains the procedure itself, and the index `Resumption`, which is the resumption point or 'location counter' for the activation. Resumption points can also be used to access and alter the execution and the source code for the procedure; details are given in Reference 2.

Since activations are just tables, entries can be added, removed, or changed as desired. For example, the `Procedure` entry need not correspond to the procedure from which the activation was created; it can be changed to any procedure. Missing local variables and parameters are created as necessary. It is typical, for example, for tables to contain *both* data and activation information. For example, a procedure to format a paper can be included as the `Procedure` entry in the table `paper`, described above. Indeed, activations can be constructed from scratch by constructing the table from the appropriate pieces as follows:

```
paper = [
    "title": "EZ Processes",
    "authors": drh || "\n" || mk,
    "body": ...,
    "references": references,
    "Procedure": formatter
]
```

The table constructor $[i_1{:}e_1, \ldots, i_n{:}e_n]$ constructs a table with n index-value pairs $(i_1, e_1), \ldots, (i_n, e_n)$. Assuming `formatter` is a procedure that formats a document defined by

```
procedure formatter(title, authors,
        body, references)
    ...
end
```

`paper()` produces the formatted document.

4. Processes

Processes in *EZ* are activations that execute concurrently. All processes share the same, persistent virtual address space. Thus, they are semantically similar to 'threads' or 'lightweight processes' in some operating systems [4]. Processes are just tables; they have all of the normal characteristics of other *EZ* values, including persistence, and all table operations apply to processes. There is no analog of 'heavyweight processes' with separate address spaces as in most operating systems. As described below, the protection benefits of such processes can be obtained by appropriate exploitation of *EZ*'s scope rules.

Processes are manipulated explicitly as *EZ* values and by a set of built-in procedures.

`create(p,` *args...*`)`

instantiates a new process. It creates a new process, passes *args* to the new process, begins its execution, and returns the process. If `p` is a table (i.e., a procedure activation) a copy is made and instantiated; if `p` is a procedure, `table(p)` is instantiated. If the process cannot be instantiated, `create` does not return a value. Other values are converted to procedures and instantiated as just described. For example, at startup, the *EZ* server (see below) executes

```
create(procedure ()
    while (1) pause()
end)
```

to instantiate an idle process. The argument is compiled into an anonymous procedure, an activation is created, and the activation is instantiated. `pause()` is a built-in procedure that causes the calling process to relinquish its processor. *EZ* uses preemptive scheduling, so `pause` is strictly unnecessary, but `pause` is useful in applications that involve waiting, such as interactive user interfaces [5, 6].

In the current implementation, *EZ* is a server. Local and remote clients connect to the server much like the remote login mechanism on UNIX. Once connected, clients initiate a 'command interpreter,' which is a single process written in *EZ*. The default is a simple line-by-line command interpreter process created by the following code.

```
create(procedure main()
    while (1) read()()
end)
```

`read()` returns a line, which is converted to a procedure (i.e., compiled) and invoked. Alternatives, which mirror the structure of the UNIX shell, execute each command as a process:

```
create(procedure main() local p
    while (1)
        if (p = create(read()))
            join(p)
end)
```

The built-in procedure `join(p)` causes the calling process to waiting until process `p` terminates.

As shown, the command interpreter process runs forever. Terminating and disconnecting a client need not terminate running processes; they continue running and the command interpreter continues running when the client reconnects. Clients can also terminate and kill all running processes. This is accomplished by explicit manipulation of the table of processes described below.

More elaborate user interfaces, such as the editor that edits *EZ* values [2], permit users to identify code fragments, perhaps by selecting them on the screen, and executing the fragments directly or as processes as suggested above. Thus, when these interfaces run as processes, they work like the simple one above differing only in their presentation to the user.

Processes terminate execution by returning or calling the built-in procedure `die()`. Note that the table remains and may be used again if it is accessible. In *EZ*, accessibility determines the lifetime of data, but not the lifetime of process execution (a garbage collector reclaims inaccessible data). For example,

```
d = create(
    procedure nextmove(x, y)
        local t = f(x, y)
        y = g(x, y)
        x = t
    end,
    0, 0)
```

creates a process for `nextmove` that, given a current position `x,y` computes a new position (perhaps a time-consuming computation) and terminates. The next move is computed by

```
d = create(d)
```

because `d` contains the values of `x` and `y` from the first instantiation.

The built-in procedure `kill(p)` terminates the execution of the process indicated by `p`, but has no affect on the accessibility of `p`.

4.1 The Process Table

Whenever a process is instantiated by `create`, a record describing the new process is entered into the table `Processes`. `Processes` is indexed by each process (i.e., each table); an entry for a process `p` is inserted by executing the equivalent of the assignment

```
Processes[p] = record [
    priority: 0,
    process: p,
    join: ,
    queue:
]
```

The `record` constructor builds a table with the indicated fields, which must be identifiers. Fields with omitted values are uninitialized. Such tables are equivalent to other tables except new entries cannot be added and existing entries cannot be removed.

The value of the `priority` field gives the execution priority of the process in the usual manner (except that all processes are guaranteed some execution time to prevent starvation). The priority of process `p` can be

changed by simply changing `Processes[p].priority`. By default, all processes are given the same priority. 'System' processes are assigned other priorities; the idle process is given a low priority while command interpreters are given a high priority. The `process` field is the process itself. The `join` field is a list of processes (threaded through the `queue` fields of the records described above) waiting for this process to terminate.

The process table can be manipulated like any other table. For example,

```
create(procedure()
    local old = 0, new
    while (1) {
        new = size(Processes)
        if (new ~= old)
            write("process count = ",
                old = new)
        pause()
    }
end)
```

instantiates a process that reports changes in the number of processes; `size` is a built-in procedure that returns the size of its operand, e.g., the number of entries in a table, the length of a string, etc.

Most operating systems have a 'process status' utility that gives information on the currently running processes; `ps` in UNIX is an example. The following procedure provides a similar facility in *EZ*.

```
procedure ps() local p
    for (p in Processes)
        write(p.Procedure, "\n")
end
```

`ps()` prints the source code for the initial procedure in each process.

4.2 Synchronization

Semaphores are used for low-level process synchronization. Higher-level synchronization mechanisms, such as events and message passing, can be implemented in *EZ* with semaphores.

`P(s)` and `V(s)` are the usual atomic P and V operations on a general semaphore `s`. A semaphore is a table with an appropriately initialized `count` entry. Processes waiting on `s` are linked together via the `queue` fields mentioned above; this list emanates from the `queue` field of `s`, which is created if necessary.

Mutual exclusion can be implemented as usual:

```
mutex = record [ count: 1, queue: ]

procedure update()
    P(mutex); { access resource }; V(mutex)
end
```

Any table can be used as a semaphore; a `count` field with a value of `0` is added if necessary. For example,

```
printer = create(procedure server()
    local head = 0, tail = 0,
        mutex = [ "count": 1 ]
    while (1) {
        P(printer)
        print(printer[head += 1])
        remove(printer, head)
    }
end)
```

creates a simple print server process and assigns it to `printer`. `printer` is a table that serves three purposes. First, it's the activation record for procedure `server`. Second, it's a semaphore that controls the producer-consumer relationship between the single process `printer` and its multiple clients. Since there's no `count` field in `printer`, one is added at the first `P(printer)` (or `V(printer)`) with the value `0`, which causes

`printer` to wait for something to print. (This implicit creation of a `count` field can be avoided by declaring a local variable `count`.) Third, `printer` is also the print queue; `printer[head+1]` through `printer[tail]` contain the values to be printed. `head` and `tail` are simply incremented; aside from integer overflow, there is no need to keep them within specific bounds.

`mutex` is a semaphore that synchronizes access to the tail of the queue, i.e., `printer[tail]`. Such synchronization is necessary because, while `printer` is the single consumer, there are many producers. As shown above, `printer` blocks on `printer` until a client executes a `V(printer)`. Clients call `lpr`, which appends a value to the queue, using `mutex` gain exclusive access, and then issues a `V` on `printer`:

```
procedure lpr(s)
    P(printer.mutex)
    printer[printer.tail += 1] = s
    V(printer.mutex)
    V(printer)
end
```

Although not recommended, semaphore `count` values and queues can be changed explicitly; they are just table entries.

4.3 Encapsulating I/O

EZ has no I/O primitives; procedures, like `read` and `write` used the examples above, do I/O by communicating with built-in processes connected to devices. These built-in processes use semaphores to synchronize producer-consumer interactions in the usual manner. Other fields are included depending on the specific device. (Of course, there are no 'disk files' as in conventional systems, *per se*, since *EZ* strings and tables provide equivalent facilities.)

For example, the value of `Input` is a built-in process that continually reads lines from the client's terminal. `Input` is initialized by the internal equivalent of

```
Input = create(
    procedure(filename)
        local line, empty, full
        empty = record [ count: 1, queue: ]
        full  = record [ count: 0, queue: ]
        while (1) {
            P(empty)
            doio
            V(full)
        }
    end,
    "terminal"
)
```

where *doio* represents the internal code that performs the actual input operations. When this process runs, *doio* reads the next line from the terminal and assigns it to `line`, and the process signals using `V` that the line is available. It then waits for a consumer to retrieve the line. Actual implementations for other devices might read more than a single line with a single I/O operation and buffer the excess.

`read` gets the next line by synchronizing appropriately and accessing `Input.line`:

```
procedure read()
    local line
    P(Input.full)
    line = Input.line
    V(Input.empty)
    return line
end
```

Terminal output is similar; the value of `Output` is a built-in process that writes the value of `Output.line` to the terminal.

Other I/O devices are accommodated by changing the details and fields of the process to suit the device. A mouse, for example, can be represented by a process with fields that encode the possible mouse events.

Uses of *EZ* 'I/O processes' thus far have been relatively simple. More ambitious applications, such as building a window system (based on NeWS or X windows) for *EZ*, are underway.

4.4 Protection

EZ provides memory protection for its processes with its scope rules, which can be used to create a 'protected' environment for a process.

Scope rules, which dictate the interpretation of free identifiers, depend on the contents of tables interrogated by the compiler. Unlike most other systems, these 'symbol tables' are *EZ* tables and can thus be manipulated by *EZ* code. As suggested above, use of identifiers declared `local` is restricted to the associated procedure in the usual manner. An interpretation of free identifiers is made by searching the table that is the current value of the variable `root` for an index value lexically equal to the identifier. Each process has its own `root`. Thus, the assignment

```
message = "I'll return soon"
```

is equivalent to

```
root.message = "I'll return soon"
```

If the identifier is not found in `root`, the compiler searches the chain of tables given by

```
root[".."]
root[".."][".."]
root[".."][".."][".."]
...
```

until the identifier is found or a table without a `".."` entry or whose `".."` entry is not a table is encountered. If this search fails to locate the identifier, it is entered in `root`.

By changing the value of `root` and altering the path given by the `".."` entries, rules such as the inheritance rules in Smalltalk [7], the 'search lists' in UNIX, and the protection benefits of separate address spaces can be obtained. When a new client connects to the *EZ* server, it executes the following initialization code.

```
for (i in root[".."])
    root[i] = root[".."][i]
root.System = root[".."]
remove(root, "..", "i")
```

This code makes a copy of the system root table, which contains the builtin values and values maintained by the server (e.g., `Processes`), arranges for `System` to refer to the system root table, and then removes the `".."` and `i` entries. Doing so prevents subsequent processes from changing system-wide values or unintentionally inferring with other clients. `System` provides an alternative path to the root table, however, for those applications with a legitimate need to manipulate system tables. Clients wishing total isolation from the server and other clients can execute

```
remove(root, "System")
```

5. Implementation

The current version of *EZ* is derived from the earlier implementation and is written in ANSI C [8] and runs under UNIX. The implementation consist of about 7,000 lines of C and a several hundred lines of *EZ*. The *EZ* code consists of the procedures described above as well as other utility and start-up procedures.

EZ source code is compiled into code for a virtual machine that mirrors the primitive operations in the language. The implementation details are similar to those in other very high-level languages, like Icon [9]; additional details are given in Reference 2.

The persistent virtual address space resides on secondary storage and is allocated in units of logical pages. To accommodate small values efficiently, logical pages are 512 bytes regardless of the physical page size of the host. All values, including tables representing activations and processes, are allocated in the virtual address space. The few necessary 'internal' structures, such as an internal pointer to the process table, are also allocated in the virtual address space.

Software caching provides access to the virtual address space. Value representations are designed to increase reference locality, which helps a large cache ($\approx$ 1000 logical pages) minimize paging. Simply flushing the cache saves the system state.

The server multiplexes a single interpreter among the running *EZ* processes initiated by the clients. This implementation is similar to implementations of 'lightweight' processes in UNIX [10, 11] and in other operating systems [4]. Execution of virtual machine instructions and of most built-in procedures is atomic. Context switching is limited to a few built-in procedures that deal with processes (e.g., `join`, `pause`, `die`) and I/O. Preemption is implemented by context switching every n ($\approx$ 1000) virtual machine instructions. I/O interrupts can also cause a context switch.

Multiple representations for values, particularly tables, are used to provide a representation that is best suited for the inferred use. For example, tables that are indexed only by integers have a different representation (sparse arrays) than those indexed by other values (hash tables). The conversion from the simpler to the more complex representation is automatic and performed only when necessary.

Most activations, including processes, are never accessed as tables, and the 'default' activation representation anticipates this pattern of 'transient' activations. While the space for all activations is logically allocated in the virtual address space, activations that are never used as tables are not written to secondary storage. Furthermore, the space for these activations is reused without incurring any paging overhead. Thus, for procedures invoked in the 'normal' functional manner, activations are allocated in a stack-like fashion as in most languages, and the allocation is nearly as efficient as stack allocation, once a pool of available activation pages accumulates.

When an activation is accessed as a table, its representation is converted to a representation that caters to both indexed access and access from the generated virtual machine code. This is accomplished by building a hash table for the indexed access and what amounts to a transfer vector that points into the hash table for access from the generated code.

To terminate the *EZ* server, all accessible activations are converted to the representation just described. This is necessary because the representation of transient activations cannot be written to secondary storage.

As described in References 2 and 12, an off-line garbage collector reclaims inaccessible pages in secondary storage representation of the virtual address space. Reclaimed pages are added to a pool of free pages for subsequent allocation. Pages from this pool are also used for transient activations.

6. Discussion

Initial use of the *EZ* process facility, though limited, has been revealing. While the persistent nature of processes is useful, it can be confusing if processes are used in the traditional, throw-away fashion. It's easy to forget that processes are just persistent values; uninitiated users are often surprised when `ps` (see above) reports days-old processes that are either blocked or computing away. More experience is necessary to understand fully the ramifications of persistent processes and to develop an methodology for their effective use.

Work is currently underway in several areas: Language issues, multi-process applications, a multiprocessor implementation, and multiple representations.

Introducing processes into *EZ* at the language level has been accompanied with some semantic problems whose 'correct' resolution remains open. One of the most interesting issues is the interaction between the process table and *EZ*'s scope rules. As described above, the compiler binds identifiers by consulting a list of symbol tables, which are just *EZ* tables. `Processes` is simply another identifier and, in each client environment, `Processes` points to the single, system-wide process table, which is also an *EZ* table. An alternative is to have many process tables and have `Processes` point to a separate table in each client. The linguistic implications of this alternative and its implementation consequences are yet to be completely investigated.

Another open question concerns 'runaway' inaccessible processes. For instance,

```
p = create(procedure ()
    while (1) pause()
end)
remove(Processes, p)
p = 0
```

creates a non-terminating process, removes it from the process table, and changes **p** so that it no longer refers to the process. The running process is inaccessible and, unless special measures are taken, will be ultimately reclaimed by the garbage collector. Currently, this situation is handled by keeping a 'hidden' pointer to all executable processes so that they *are* accessible, but not from the source-language level. An alternative under investigation would have the garbage collector terminate such processes. More experience may suggest the 'right' choice or other alternatives.

Applications of *EZ* processes thus far have been relatively simple; more ambitious applications will contribute understanding and help identify problems in the current implementation. An example is a current project to use the X window system in the *EZ* clients. Not only will this application provide a more usable system, but will help refine the mechanisms used to have *EZ* processes respond to interrupts. It will also require I/O processes that deal with bitmapped displays.

The current implementation runs on a uniprocessor UNIX system. An implementation for a recently acquired 4-processor DEC 'Firefly' is also planned. The Firefly's shared memory architecture is matched perfectly to *EZ*'s abstract view of a single large, flat address space. A multi-processor implementation will also test the suitability of the process mechanism and its implementation. For example, the decision to make virtual machine instructions atomic helped evolve the implementation of 'sequential' *EZ* to include processes, but is inappropriate on a multiprocessor. Significant redesign of the virtual machine will be required to deal with 'big' operations, such as compilation and concatenation of long strings, that simply cannot be atomic in a multiprocessor system.

Implementing *EZ* tends to require specialized techniques because the semantics are so different from traditional languages and operating systems. While all implementations to date have used multiple representations, processes stress the current implementation to its limit. New representations for processes are being investigated; the approach taken for transient activations, mentioned in the previous section, is an example. Measurements to confirm the efficacy of alternative representations are also contemplated.

Implementing and using *EZ*'s processes has also suggested numerous areas for future work. Examples include a concurrent garbage collector as an *EZ* process [13], performance monitoring for persistent processes, and long-term reliability mechanisms. A particularly interesting area is in building a 'distributed' *EZ*, perhaps by adapting the recent implementations of shared virtual memory [14, 15] to, in effect, distribute the *EZ* virtual address space across a network of workstations.

Acknowledgements

Chris Fraser's comments helped clarify several sections of this paper.

References

1. C. W. Fraser and D. R. Hanson, A High-Level Programming and Command Language, *Proc. of the SIGPLAN '83 Symp. on Programming Language Issues in Software Systems*, San Francisco, CA, June 1983, 212–219.
2. C. W. Fraser and D. R. Hanson, High-Level Language Facilities for Low-Level Services, *Conf. Rec. 12th ACM Symp. on Prin. of Programming Languages*, New Orleans, LA, Jan. 1985, 217–224.
3. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1983.
4. M. J. Accetta, R. V. Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, Jr., and M. W. Young, Mach: A New Kernel Foundation for UNIX Development, *Proc. of the Summer USENIX Conf.*, Atlanta, GA, July 1986, 93–112.
5. *NeWS 1.1 Manual*, Sun Microsystems Inc., Mountain View, CA, Jan. 1988.
6. *NeWS Technical Overview*, Sun Microsystems Inc., Mountain View, CA, Jan. 1988.
7. A. Goldberg, D. Robson, and D. H. H. Ingalls, *Smalltalk-80: The Language and Its Implementation*, Addison Wesley, Reading, MA, 1983.
8. B. W. Kernighan and D. M. Ritchie, *The C Programming Language, Second Edition*, Prentice-Hall, Englewood Cliffs, NJ, Second Edition, 1988.

9. R. E. Griswold and M. T. Griswold, *The Implementation of the Icon Programming Language*, Princeton Univ. Press, Princeton, NJ, 1986.
10. E. C. Cooper and R. P. Draves, C Threads, Tech. Rep. CMU-CS-88–154, Computer Science Dept., Carnegie Mellon Univ., Pittsburgh, PA, June 1988.
11. G. V. Cormack, A Micro-Kernel for Concurrency in C, *Software—Practice & Experience* **18**, 5 (May 1988) 485–492.
12. C. W. Fraser and D. R. Hanson, The *EZ* Reference Manual, Tech. Rep. 84–1, Dept. of Computer Science, The Univ. of Arizona, Tucson, AZ, Jan. 1984.
13. A. W. Appel, J. R. Ellis, and K. Li, Real-time Concurrent Collection on Stock Multiprocessors, *Proc. SIGPLAN '88 Conf. on Programming Language Design and Implementation*, Atlanta, GA, June 1988, 11–20.
14. K. Li, *Shared Virtual Memory on Loosely-Coupled Multiprocessors*, Ph.D. Diss., Yale Univ., New Haven, CT, Oct. 1986.
15. K. Li and P. Hudak, Memory Coherence in Shared Virtual Memory Systems, *Proc. 5th Annual Symp. on Principles of Distributed Computing*, Calgary, Alberta, Aug. 1986, 229–239.