Concepts and Notations for Concurrent Programming

GREGORY R. ANDREWS

Department of Computer Science, University of Arizona, Tucson, Arizona 85721

FRED B. SCHNEIDER

Department of Computer Science, Cornell University, Ithaca, New York 14853

Much has been learned in the last decade about concurrent programming. This paper identifies the major concepts of concurrent programming and describes some of the more important language notations for writing concurrent programs. The roles of processes, communication, and synchronization are discussed. Language notations for expressing concurrent execution and for specifying process interaction are surveyed. Synchronization primitives based on shared variables and on message passing are described. Finally, three general classes of concurrent programming languages are identified and compared.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.3 [**Programming Languages**]: Language Constructs—*concurrent programming structures*; *coroutines*; D.4.1 [**Operating Systems**]: Process Management; D.4.7 [**Operating Systems**]: Organization and Design

General Terms: Algorithms, Languages

INTRODUCTION

The complexion of concurrent programming has changed substantially in the past ten years. First, theoretical advances have prompted the definition of new programming notations that express concurrent computations simply, make synchronization requirements explicit, and facilitate formal correctness proofs. Second, the availability of inexpensive processors has made possible the construction of distributed systems and multiprocessors that were previously economically infeasible. Because of these two developments, concurrent programming no longer is the sole province of those who design and implement operating systems; it has become important to programmers of all kinds of applications, including database management systems, large-scale parallel scientific computations, and real-time, embedded control systems. In fact, the discipline has matured to the point that there are now undergraduate-level text books devoted solely to the topic [Holt et al., 1978; Ben-Ari, 1982]. In light of this growing range of applicability, it seems appropriate to survey the state of the art.

This paper describes the concepts central to the design and construction of concurrent programs and explores notations for describing concurrent computations. Although this description requires detailed discussions of some concurrent programming languages, we restrict attention to those whose designs we believe to be influential or conceptually innovative. Not all the languages we discuss enjoy widespread use. Many are experimental efforts that focus on understanding the interactions of a given collection of constructs. Some have not even been implemented; others have been, but with little concern for efficiency, access control, data types, and other important (though nonconcurrency) issues.

We proceed as follows. In Section 1 we

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1983 ACM 0010-4892/83/0300-0003 $00.75

CONTENTS

INTRODUCTION
1. CONCURRENT PROGRAMS: PROCESSES AND PROCESS INTERACTION
   1.1 Processes
   1.2 Process Interaction
2. SPECIFYING CONCURRENT EXECUTION
   2.1 Coroutines
   2.2 The **fork** and **join** Statements
   2.3 The **cobegin** Statement
   2.4 Process Declarations
3. SYNCHRONIZATION PRIMITIVES BASED ON SHARED VARIABLES
   3.1 Busy-Waiting
   3.2 Semaphores
   3.3 Conditional Critical Regions
   3.4 Monitors
   3.5 Path Expressions
4. SYNCHRONIZATION PRIMITIVES BASED ON MESSAGE PASSING
   4.1 Specifying Channels of Communication
   4.2 Synchronization
   4.3 Higher Level Message-Passing Constructs
   4.4 An Axiomatic View of Message Passing
   4.5 Programming Notations Based on Message Passing
5. MODELS OF CONCURRENT PROGRAMMING LANGUAGES
6. CONCLUSION
ACKNOWLEDGMENTS
REFERENCES

discuss the three issues that underlie all concurrent programming notations: how to express concurrent execution, how processes communicate, and how processes synchronize. These issues are treated in detail in the remainder of the paper. In Section 2 we take a closer look at various ways to specify concurrent execution: coroutines, **fork** and **cobegin** statements, and **process** declarations. In Section 3 we discuss synchronization primitives that are used when communication uses shared variables. Two general types of synchronization are considered—exclusion and condition synchronization—and a variety of ways to implement them are described: busy-waiting, semaphores, conditional critical regions, monitors, and path expressions. In Section 4 we discuss message-passing primitives. We describe methods for specifying channels of communication and for synchronization, and higher level constructs for performing remote procedure calls and atomic transactions. In Section 5 we identify and compare three general classes of concurrent programming languages. Finally, in Section 6, we summarize the major topics and identify directions in which the field is headed.

1. CONCURRENT PROGRAMS: PROCESSES AND PROCESS INTERACTION

1.1 Processes

A *sequential program* specifies sequential execution of a list of statements; its execution is called a *process*. A *concurrent program* specifies two or more sequential programs that may be executed concurrently as *parallel processes*. For example, an airline reservation system that involves processing transactions from many terminals has a natural specification as a concurrent program in which each terminal is controlled by its own sequential process. Even when processes are not executed simultaneously, it is often easier to structure a system as a collection of cooperating sequential processes rather than as a single sequential program. A simple batch operating system can be viewed as three processes: a *reader* process, an *executer* process, and a *printer* process. The *reader* process reads cards from a card reader and places card images in an input buffer. The *executer* process reads card images from the input buffer, performs the specified computation (perhaps generating line images), and stores the results in an output buffer. The *printer* process retrieves line images from the output buffer and writes them to a printer.

A concurrent program can be executed either by allowing processes to share one or more processors or by running each process on its own processor. The first approach is referred to as *multiprogramming*; it is supported by an operating system kernel [Dijkstra, 1968a] that multiplexes the processes on the processor(s). The second approach is referred to as *multiprocessing* if the processors share a common memory (as in a multiprocessor [Jones and Schwarz, 1980]), or as *distributed processing* if the processors are connected by a communica-

tions network.[1] Hybrid approaches also exist—for example, processors in a distributed system are often multiprogrammed.

The rate at which processes are executed depends on which approach is used. When each process is executed on its own processor, each is executed at a fixed, but perhaps unknown, rate; when processes share a processor, it is as if each is executed on a variable-speed processor. Because we would like to be able to understand a concurrent program in terms of its component sequential processes and their interaction, without regard for how they are executed, we make no assumption about execution rates of concurrently executing processes, except that they all are positive. This is called the *finite progress assumption.* The correctness of a program for which only finite progress is assumed is thus independent of whether that program is executed on multiple processors or on a single multiprogrammed processor.

1.2 Process Interaction

In order to cooperate, concurrently executing processes must communicate and synchronize. Communication allows execution of one process to influence execution of another. Interprocess communication is based on the use of shared variables (variables that can be referenced by more than one process) or on message passing.

Synchronization is often necessary when processes communicate. Processes are executed with unpredictable speeds. Yet, to communicate, one process must perform some action that the other detects—an action such as setting the value of a variable or sending a message. This only works if the events "perform an action" and "detect an action" are constrained to happen in that order. Thus one can view synchronization as a set of constraints on the ordering of events. The programmer employs a *synchronization mechanism* to delay execution of a process in order to satisfy such constraints.

To make the concept of synchronization a bit more concrete, consider the batch operating system described above. A shared buffer is used for communication between the *reader* process and the *executer* process. These processes must be synchronized so that, for example, the *executer* process never attempts to read a card image from the input if the buffer is empty.

This view of synchronization follows from taking an *operational approach* to program semantics. An execution of a concurrent program can be viewed as a sequence of *atomic actions,* each resulting from the execution of an indivisible operation.[2] This sequence will comprise some interleaving of the sequences of atomic actions generated by the individual component processes. Rarely do all execution interleavings result in acceptable program behavior, as is illustrated in the following. Suppose initially that $x = 0$, that process *P1* increments x by 1, and that process *P2* increments x by 2:

P1: $x := x + 1$ *P2*: $x := x + 2$

It would seem reasonable to expect the final value of x, after *P1* and *P2* have executed concurrently, to be 3. Unfortunately, this will not always be the case, because assignment statements are not generally implemented as indivisible operations. For example, the above assignments might be implemented as a sequence of three indivisible operations: (i) load a register with the value of x; (ii) add 1 or 2 to it; and (iii) store the result in x. Thus, in the program above, the final value of x might be 1, 2, or 3. This anomalous behavior can be avoided by preventing interleaved execution of the two assignment statements—that is, by controlling the ordering of the events corresponding to the atomic actions. (If ordering were thus controlled, each assignment statement would be an indivisible operation.) In other words, execution of *P1* and *P2* must be synchronized by enforcing restrictions on possible interleavings.

The *axiomatic approach* [Floyd, 1967; Hoare, 1969; Dijkstra, 1976] provides a sec-

[1] A concurrent program that is executed in this way is often called a *distributed program.*

[2] We assume that a single memory reference is indivisible; if two processes attempt to reference the same memory cell at the same time, the result is as if the references were made serially. This is a reasonable assumption in light of the way memory is constructed. See Lamport [1980b] for a discussion of some of the implications of relaxing this assumption.

ond framework in which to view the role of synchronization.[3] In this approach, the semantics of statements are defined by axioms and inference rules. This results in a formal logical system, called a "programming logic." Theorems in the logic have the form

$$\{P\}\ S\ \{Q\}$$

and specify a relation between statements (S) and two predicates, a *precondition* P and a *postcondition* Q. The axioms and inference rules are chosen so that theorems have the interpretation that if execution of S is started in any state that satisfies the precondition, and if execution terminates, then the postcondition will be true of the resulting state. This allows statements to be viewed as relations between predicates.

A *proof outline*[4] provides one way to present a program and its proof. It consists of the program text interleaved with assertions so that for each statement S, the triple (formed from (1) the assertion that textually precedes S in the proof outline, (2) the statement S, and (3) the assertion that textually follows S in the proof outline) is a theorem in the programming logic. Thus the appearance of an assertion R in the proof outline signifies that R is true of the program state when control reaches that point.

When concurrent execution is possible, the proof of a sequential process is valid only if concurrent execution of other processes cannot invalidate assertions that appear in the proof [Ashcroft, 1975; Keller, 1976; Owicki and Gries, 1976a, 1976b; Lamport, 1977, 1980a; Lamport and Schneider, 1982]. One way to establish this is to assume that the code between any two assertions in a proof outline is executed atomically[5] and then to prove a series of theorems showing that no statement in one process invalidates any assertion in the proof of another. These additional theorems constitute a proof of *noninterference*. To illustrate this, consider the following excerpt from a proof outline of two concurrent processes *P1* and *P2*:

```
P1:   ...                    P2:   ...
      {x > 0}                      {x < 0}
   S1: x := 16                  S2: x := -2
      {x = 16}                     ...
      ...
```

In order to prove that execution of *P2* does not interfere with the proof of *P1*, part of what we must show is that execution of *S2* does not invalidate assertions $\{x > 0\}$ and $\{x = 16\}$ in the proof of *P1*. This is done by proving

$$\{x < 0 \textbf{ and } x > 0\}\ x := -2\ \{x > 0\}$$

and

$$\{x < 0 \textbf{ and } x > 0\}\ x := -2\ \{x = 16\}$$

Both of these are theorems because the precondition of each, $\{x < 0 \textbf{ and } x > 0\}$, is false. What we have shown is that execution of *S2* is not possible when either the precondition or postcondition of *S1* holds (and thus *S1* and *S2* are mutually exclusive). Hence, *S2* cannot invalidate either of these assertions.

Synchronization mechanisms control interference in two ways. First, they can delay execution of a process until a given condition (assertion) is true. By so doing, they ensure that the precondition of the subsequent statement is guaranteed to be true (provided that the assertion is not interfered with). Second, a synchronization mechanism can be used to ensure that a block of statements is an indivisible operation. This eliminates the possibility of statements in other processes interfering with assertions appearing within the proof of that block of statements.

Both views of programs, operational and axiomatic, are useful. The operational approach—viewing synchronization as an ordering of events—is well suited to explaining how synchronization mechanisms work. For that reason, the operational approach is used rather extensively in this survey. It also constitutes the philosophical basis for a family of synchronization mechanisms called *path expressions* [Campbell and Ha-

[3] We include brief discussions of axiomatic semantics here and elsewhere in the paper because of its importance in helping to explain concepts. However, a full discussion of the semantics of concurrent computation is beyond the scope of this paper.

[4] This sometimes is called an asserted program.

[5] This should be construed as specifying what assertions must be included in the proof rather than as a restriction on how statements are actually executed.

bermann, 1974], which are described in Section 3.5.

Unfortunately, the operational approach does not really help one understand the behavior of a concurrent program or argue convincingly about its correctness. Although it has borne fruit for simple concurrent programs—such as transactions processed concurrently in a database system [Bernstein and Goodman, 1981]—the operational approach has only limited utility when applied to more complex concurrent programs [Akkoyunlu et al., 1978; Bernstein and Schneider, 1978]. This limitation exists because the number of interleavings that must be considered grows exponentially with the size of the component sequential processes. Human minds are not good at such extensive case analysis. The axiomatic approach usually does not have this difficulty. It is perhaps the most promising technique for understanding concurrent programs. Some familiarity with formal logic is required for is use, however, and this has slowed its acceptance.

To summarize, there are three main issues underlying the design of a notation for expressing a concurrent computation:

(i) how to indicate concurrent execution;
(ii) which mode of interprocess communication to use;
(iii) which synchronization mechanism to use.

Also, synchronization mechanisms can be viewed either as constraining the ordering of events or as controlling interference. We consider all these topics in depth in the remainder of the paper.

2. SPECIFYING CONCURRENT EXECUTION

Various notations have been proposed for specifying concurrent execution. Early proposals, such as the **fork** statement, are marred by a failure to separate process definition from process synchronization. Later proposals separate these distinct concepts and characteristically possess syntactic restrictions that impose some structure on a concurrent program. This structure allows easy identification of those program segments that can be executed concurrently. Consequently, such proposals are well suited for use with the axiomatic approach, because the structure of the program itself clarifies the proof obligations for establishing noninterference.

Below, we describe some representative constructs for expressing concurrent execution. Each can be used to specify computations having a *static* (fixed) number of processes, or can be used in combination with process-creation mechanisms to specify computations having a *dynamic* (variable) number of processes.

2.1 Coroutines

Coroutines are like subroutines, but allow transfer of control in a symmetric rather than strictly hierarchical way [Conway, 1963a]. Control is transferred between coroutines by means of the **resume** statement. Execution of **resume** is like execution of procedure **call**: it transfers control to the named routine, saving enough state information for control to return later to the instruction following the **resume**. (When a routine is first resumed, control is transferred to the beginning of that routine.) However, control is returned to the original routine by executing another **resume** rather than by executing a procedure **return**. Moreover, any other coroutine can potentially transfer control back to the original routine. (For example, coroutine *C1* could **resume** *C2*, which could **resume** *C3*, which could **resume** *C1*.) Thus **resume** serves as the only way to transfer control between coroutines, and one coroutine can transfer control to any other coroutine that it chooses.

A use of coroutines appears in Figure 1. Note that **resume** is used to transfer control between coroutines *A* and *B*, a **call** is used to initiate the coroutine computation, and **return** is used to transfer control back to the caller *P*. The arrows in Figure 1 indicate the transfers of control.

Each coroutine can be viewed as implementing a process. Execution of **resume** causes process sychronization. When used with care, coroutines are an acceptable way to organize concurrent programs that share a single processor. In fact, multiprogramming can also be implemented using coroutines. Coroutines are not adequate for true parallel processing, however, because their semantics allow for execution of only

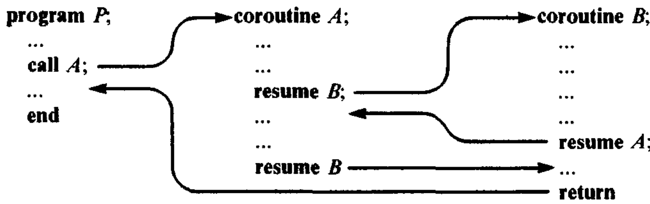

Figure 1. A use of coroutines.

one routine at a time. In essence, coroutines are concurrent processes in which process switching has been completely specified, rather than left to the discretion of the implementation.

Statements to implement coroutines have been included in discrete event simulation languages such as SIMULA I [Nygaard and Dahl, 1978] and its successors; the string-processing language SL5 [Hanson and Griswold, 1978]; and systems implementation languages including BLISS [Wulf et al., 1971] and most recently Modula-2 [Wirth, 1982].

2.2 The fork and join Statements

The **fork** statement [Dennis and Van Horn, 1966; Conway, 1963b], like a **call** or **resume**, specifies that a designated routine should start executing. However, the invoking routine and the invoked routine proceed concurrently. To synchronize with completion of the invoked routine, the invoking routine can execute a **join** statement. Executing **join** delays the invoking routine until the designated invoked routine has terminated. (The latter routine is often designated by a value returned from execution of a prior **fork**.) A use of **fork** and **join** follows:

```
program P1;          program P2;
  ...                  ...
  fork P2;             ...
  ...                  ...
  join P2;             end
  ...
```

Execution of *P2* is initiated when the **fork** in *P1* is executed; *P1* and *P2* then execute concurrently until either *P1* executes the **join** statement or *P2* terminates. After *P1* reaches the **join** and *P2* terminates, *P1* executes the statements following the **join**.

Because **fork** and **join** can appear in conditionals and loops, a detailed understanding of program execution is necessary to understand which routines will be executed concurrently. Nevertheless, when used in a disciplined manner, the statements are practical and powerful. For example, **fork** provides a direct mechanism for dynamic process creation, including multiple activations of the same program text. The UNIX[6] operating system [Ritchie and Thompson, 1974] makes extensive use of variants of **fork** and **join**. Similar statements have also been included in PL/I and Mesa [Mitchell et al., 1979].

2.3 The cobegin Statement

The **cobegin** statement[7] is a structured way of denoting concurrent execution of a set of statements. Execution of

cobegin $S_1 \parallel S_2 \parallel \cdots \parallel S_n$ **coend**

causes concurrent execution of $S_1, S_2, \ldots, S_n$. Each of the S_i's may be any statement, including a **cobegin** or a block with local declarations. Execution of a **cobegin** statement terminates only when execution of all the S_i's have terminated.

Although **cobegin** is not as powerful as **fork/join**,[8] it is sufficient for specifying

[6] UNIX is a trademark of Bell Laboratories.

[7] This was first called **parbegin** by Dijkstra [1968b].

[8] Execution of a concurrent program can be represented by a *process flow graph*: an acyclic, directed graph having one node for each process and an arc from one node to another if the second cannot execute until the first has terminated [Shaw, 1974]. Without introducing extra processes or idle time, **cobegin** and sequencing can only represent series-parallel (properly nested) process flow graphs. Using **fork** and **join**, the computation represented by any process flow graph can be specified directly. Furthermore, **fork** can be used to create an arbitrary number of concurrent processes, whereas **cobegin** as defined in any existing language, can be used only to activate a fixed number of processes.

```
program OPSYS;

  var input_buffer : array [0..N-1] of cardimage;
      output_buffer : array [0..N-1] of lineimage;

  process reader;
    var card : cardimage;
    loop
      read card from cardreader;
      deposit card in input_buffer
      end
    end;

  process executer;
    var card : cardimage;
        line : lineimage;
    loop
      fetch card from input_buffer;
      process card and generate line;
      deposit line in output_buffer
      end
    end;

  process printer;
    var line : lineimage;
    loop
      fetch line from output_buffer;
      print line on lineprinter
      end
    end

end.
```

Figure 2. Outline of batch operating system.

most concurrent computations. Furthermore, the syntax of the **cobegin** statement makes explicit which routines are executed concurrently, and provides a single-entry, single-exit control structure. This allows the state transformation implemented by a **cobegin** to be understood by itself, and then to be used to understand the program in which it appears.

Variants of **cobegin** have been included in ALGOL68 [van Wijngaarden et al., 1975], Communicating Sequential Processes [Hoare, 1978], Edison [Brinch Hansen, 1981], and Argus [Liskov and Scheifler, 1982].

2.4 Process Declarations

Large programs are often structured as a collection of sequential routines, which are executed concurrently. Although such routines could be declared as procedures and activated by means of **cobegin** or **fork**, the structure of a concurrent program is much clearer if the declaration of a routine states whether it will be executed concurrently. The *process declaration* provides such a facility.

Use of process declarations to structure a concurrent program is illustrated in Figure 2, which outlines the batch operating system described earlier. We shall use this notation for process declarations in the remainder of this paper to denote collections of routines that are executed concurrently.

In some concurrent programming languages (e.g., Distributed Processes [Brinch Hansen, 1978] and SR [Andrews, 1981]), a collection of process declarations is equivalent to a single **cobegin**, where each of the declared processes is a component of the **cobegin**. This means there is exactly one instance of each declared process. Alternatively, some languages provide an explicit mechanism—**fork** or something similar—for activating instances of process declarations. This explicit activation mechanism can only be used during program initialization in some languages (e.g., Concurrent PASCAL [Brinch Hansen, 1975] and Modula [Wirth, 1977a]). This leads to a fixed number of processes but allows mul-

tiple instances of each declared process to be created. By contrast, in other languages (e.g., PLITS [Feldman, 1979] and Ada [U. S. Department of Defense, 1981]) processes can be created at any time during execution, which makes possible computations having a variable number of processes.

3. SYNCHRONIZATION PRIMITIVES BASED ON SHARED VARIABLES

When shared variables are used for interprocess communication, two types of synchronization are useful: mutual exclusion and condition synchronization. *Mutual exclusion* ensures that a sequence of statements is treated as an indivisible operation. Consider, for example, a complex data structure manipulated by means of operations implemented as sequences of statements. If processes concurrently perform operations on the same shared data object, then unintended results might occur. (This was illustrated earlier where the statement $x := x + 1$ had to be executed indivisibly for a meaningful computation to result.) A sequence of statements that must appear to be executed as an indivisible operation is called a *critical section*. The term "mutual exclusion" refers to mutually exclusive execution of critical sections. Notice that the effects of execution interleavings are visible only if two computations access shared varibles. If such is the case, one computation can see intermediate results produced by incomplete execution of the other. If two routines have no variables in common, then their execution need not be mutually exclusive.

Another situation in which it is necessary to coordinate execution of concurrent processes occurs when a shared data object is in a state inappropriate for executing a particular operation. Any process attempting such an operation should be delayed until the state of the data object (i.e., the values of the variables that comprise the object) changes as a result of other processes executing operations. We shall call this type of synchronization *condition synchronization*.[9] Examples of condition synchronization appear in the simple batch operating system discussed above. A process attempting to execute a "deposit" operation on a buffer (the buffer being a shared data object) should be delayed if the buffer has no space. Similarly, a process attempting to "fetch" from a buffer should be delayed if there is nothing in the buffer to remove.

Below, we survey various mechanisms for implementing these two types of synchronization.

3.1 Busy-Waiting

One way to implement synchronization is to have processes set and test shared variables. This approach works reasonably well for implementing condition synchronization, but not for implementing mutual exclusion, as will be seen. To signal a condition, a process sets the value of a shared variable; to wait for that condition, a process repeatedly tests the variable until it is found to have a desired value. Because a process waiting for a condition must repeatedly test the shared variable, this technique to delay a process is called *busy-waiting* and the process is said to be *spinning*. Variables that are used in this way are sometimes called *spin locks*.

To implement mutual exclusion using busy-waiting, statements that signal and wait for conditions are combined into carefully constructed protocols. Below, we present Peterson's solution to the two-process mutual exclusion problem [Peterson, 1981]. (This solution is simpler than the solution proposed by Dekker [Shaw, 1974].) The solution involves an *entry protocol*, which a process executes before entering its critical section, and an *exit protocol*, which a process executes after finishing its critical section:

```
process P1;
  loop
    Entry Protocol;
    Critical Section;
    Exit Protocol;
    Noncritical Section
    end
  end
process P2;
  loop
    Entry Protocol;
    Critical Section;
    Exit Protocol;
    Noncritical Section
    end
  end
```

[9] Unfortunately, there is no commonly agreed upon term for this.

Three shared variables are used as follows to realize the desired synchronization. Boolean variable *enteri* ($i = 1$ or 2) is true when process *Pi* is executing its entry protocol or its critical section. Variable *turn* records the name of the next process to be granted entry into its own critical section; *turn* is used when both processes execute their respective entry protocols at about the same time. The solution is

```
program Mutex_Example;
  var enter1, enter2 : Boolean initial (false,false);
      turn : integer initial ("P1");    { or "P2" }

  process P1;
    loop
      Entry_Protocol:
        enter1 := true;   { announce intent to enter }
        turn := "P2";   { set priority to other process }
        while enter2 and turn = "P2"
          do skip; { wait if other process is in and it is his turn }
      Critical Section;
      Exit_Protocol:
        enter1 := false;   { renounce intent to enter }
      Noncritical Section
      end
    end;

  process P2;
    loop
      Entry_Protocol:
        enter2 := true;   { announce intent to enter }
        turn := "P1";   { set priority to other process }
        while enter1 and turn = "P1"
          do skip; { wait if other process is in and it is his turn }
      Critical Section;
      Exit_Protocol:
        enter1 := false;   { renounce intent to enter }
      Noncritical Section
      end
    end
end.
```

In addition to implementing mutual exclusion, this solution has two other desirable properties. First, it is *deadlock free*. *Deadlock* is a state of affairs in which two or more processes are waiting for events that will never occur. Above, deadlock could occur if each process could spin forever in its entry protocol; using *turn* precludes deadlock. The second desirable property is *fairness*:[10] if a process is trying to enter its critical section, it will eventually be able to do so, provided that the other process exits its critical section. Fairness is a desirable property for a synchronization mechanism because its presence ensures that the finite progress assumption is not invalidated by delays due to synchronization. In general, a synchronization mechanism is *fair* if no process is delayed forever, waiting for a condition that occurs infinitely often; it is *bounded fair* if there exists an upper bound on how long a process will be delayed waiting for a condition that occurs infinitely often. The above protocol is bounded fair, since a process waiting to enter its critical section is delayed for at most one execution of the other process' critical section; the variable *turn* ensures this. Peterson [1981] gives operational proofs of mutual exclusion, deadlock freedom, and fairness; Dijkstra [1981a] gives axiomatic ones.

Synchronization protocols that use only busy-waiting are difficult to design, understand, and prove correct. First, although instructions that make two memory references part of a single indivisible operation (e.g., the TS (test-and-set) instruction on the IBM 360/370 processors) help, such instructions do not significantly simplify the task of designing synchronization protocols. Second, busy-waiting wastes processor cycles. A processor executing a spinning process could usually be employed more productively by running other processes until the awaited condition occurs. Last, the busy-waiting approach to synchronization burdens the programmer with deciding both what synchronization is required and how to provide it. In reading a program that uses busy-waiting, it may not be clear to the reader which program variables are used for implementing synchronization and which are used for, say, interprocess communication.

3.2 Semaphores

Dijkstra was one of the first to appreciate the difficulties of using low-level mechanisms for process synchronization, and this prompted his development of semaphores [Dijkstra, 1968a, 1968b]. A *semaphore* is a nonnegative integer-valued variable on which two operations are defined: **P** and **V**. Given a semaphore s, **P**(s) delays until $s > 0$ and then executes $s := s - 1$; the test and decrement are executed as an indivisible operation. **V**(s) executes $s := s + 1$ as

[10] A more complete discussion of fairness appears in Lehmann et al. [1981].

an indivisible operation.[11] Most semaphore implementations are assumed to exhibit fairness: no process delayed while executing **P**(s) will remain delayed forever if **V**(s) operations are performed infinitely often. The need for fairness arises when a number of processes are simultaneously delayed, all attempting to execute a **P** operation on the same semaphore. Clearly, the implementation must choose which one will be allowed to proceed when a **V** is ultimately performed. A simple way to ensure fairness is to awaken processes in the order in which they were delayed.

Semaphores are a very general tool for solving synchronization problems. To implement a solution to the mutual exclusion problem, each critical section is preceded by a **P** operation and followed by a **V** operation on the same semaphore. All mutually exclusive critical sections use the same semaphore, which is initialized to one. Because such a semaphore only takes on the values zero and one, it is often called a *binary* semaphore.

To implement condition synchronization, shared variables are used to represent the condition, and a semaphore associated with the condition is used to accomplish the synchronization. After a process has made the condition true, it signals that it has done so by executing a **V** operation; a process delays until a condition is true by executing a **P** operation. A semaphore that can take any nonnegative value is called a *general* or *counting* semaphore. General semaphores are often used for condition synchronization when controlling resource allocation. Such a semaphore has as its initial value the initial number of units of the resource; a **P** is used to delay a process until a free resource unit is available; **V** is executed when a unit of the resource is returned. Binary semaphores are sufficient for some types of condition synchronization, notably those in which a resource has only one unit.

A few examples will illustrate uses of semaphores. We show a solution to the two-process mutual exclusion problem in terms of semaphores in the following:

```
program Mutex_Example;

  var mutex : semaphore initial (1);

  process P1;
    loop
      P(mutex);          { Entry Protocol }
      Critical Section;
      V(mutex);          { Exit Protocol }
      Noncritical Section
      end
    end;

  process P2;
    loop
      P(mutex);          { Entry Protocol }
      Critical Section;
      V(mutex);          { Exit Protocol }
      Noncritical Section
      end
    end

  end.
```

Notice how simple and symmetric the entry and exit protocols are in this solution to the mutual exclusion problem. In particular, this use of **P** and **V** ensures both mutual exclusion and absence of deadlock. Also, if the semaphore implementation is fair and both processes always exit their critical sections, each process eventually gets to enter its critical section.

Semaphores can also be used to solve *selective mutual exclusion* problems. In the latter, shared variables are partitioned into disjoint sets. A semaphore is associated with each set and used in the same way as *mutex* above to control access to the variables in that set. Critical sections that reference variables in the same set execute with mutual exclusion, but critical sections that reference variables in different sets execute concurrently. However, if two or more processes require simultaneous access to variables in two or more sets, the programmer must take care or deadlock could result. Suppose that two processes, *P1* and *P2*, each require simultaneous access to sets of shared variables *A* and *B*. Then, *P1* and

[11] P is the first letter of the Dutch word "passeren," which means "to pass"; V is the first letter of "vrygeven," the Dutch word for "to release" [Dijkstra, 1981b]. Reflecting on the definitions of P and V, Dijkstra and his group observed the P might better stand for "prolagen" formed from the Dutch words "proberen" (meaning "to try") and "verlagen" (meaning "to decrease") and V for the Dutch word "verhogen" meaning "to increase." Some authors use **wait** for P and **signal** for V.

P2 will deadlock if, for example, *P1* acquires access to set *A*, *P2* acquires access to set *B*, and then both processes try to acquire access to the set that they do not yet have. Deadlock is avoided here (and in general) if processes first try to acquire access to the same set (e.g., *A*), and then try to acquire access to the other (e.g., *B*).

Figure 3 shows how semaphores can be used for selective mutual exclusion and condition synchronization in an implementation of our simple example operating system. Semaphore *in_mutex* is used to implement mutually exclusive access to *input_buffer* and *out_mutex* is used to implement mutually exclusive access to *output_buffer*.[12] Because the buffers are disjoint, it is possible for operations on *input_buffer* and *output_buffer* to proceed concurrently. Semaphores *num_cards*, *num_lines*, *free_cards*, and *free_lines* are used for condition synchronization: *num_cards* (*num_lines*) is the number of card images (line images) that have been deposited but not yet fetched from *input_buffer* (*output_buffer*); *free_cards* (*free_lines*) is the number of free slots in *input_buffer* (*output_buffer*). Executing **P**(*num_cards*) delays a process until there is a card in *input_buffer*; **P**(*free_cards*) delays its invoker until there is space to insert a card in *input_buffer*. Semaphores *num_lines* and *free_lines* play corresponding roles with respect to *output_buffer*. Note that before accessing a buffer, each process first waits for the condition required for access and then acquires exclusive access to the buffer. If this were not the case, deadlock could result. (The order in which **V** operations are performed after the buffer is accessed is not critical.)

Semaphores can be implemented by using busy-waiting. More commonly, however, they are implemented by system calls to a kernel. A *kernel* (sometimes called a *supervisor* or *nucleus*) implements processes on a processor [Dijkstra, 1968a; Shaw, 1974]. At all times, each process is either *ready* to execute on the processor or is *blocked*, waiting to complete a **P** operation. The kernel maintains a *ready list*—a queue of descriptors for ready processes—and multiplexes the processor among these processes, running each process for some period of time. Descriptors for processes that are blocked on a semaphore are stored on a queue associated with that semaphore; they are not stored on the ready list, and hence the processes will not be executed. Execution of a **P** or **V** operation causes a trap to a kernel routine. For a **P** operation, if the semaphore is positive, it is decre-

[12] In this solution, careful implementation of the operations on the buffers obviates the need for semaphores *in_mutex* and *out_mutex*. The semaphores that implement condition synchronization are sufficient to ensure mutually exclusive access to individual buffer slots.

```
program OPSYS;
 var in_mutex, out_mutex : semaphore initial (1,1);
     num_cards, num_lines : semaphore initial (0,0);
     free_cards, free_lines : semaphore initial (N,N);
     input_buffer : array [0..N-1] of cardimage;
     output_buffer : array [0..N-1] of lineimage;

 process reader;
   var card : cardimage;
   loop
     read card from cardreader;
     P(free_cards); P(in_mutex);
       deposit card in input_buffer;
     V(in_mutex); V(num_cards)
     end
   end;

 process executer;
   var card : cardimage;
       line : lineimage;
   loop
     P(num_cards); P(in_mutex);
       fetch card from input_buffer;
     V(in_mutex); V(free_cards);
     process card and generate line;
     P(free_lines); P(out_mutex);
       deposit line in output_buffer;
     V(out_mutex); V(num_lines)
     end
   end;

 process printer;
   var line : lineimage;
   loop
     P(num_lines); P(out_mutex);
       fetch line from output_buffer;
     V(out_mutex); V(free_lines);
     print line on lineprinter
     end
   end

 end.
```

Figure 3. Batch operating system with semaphores.

mented; otherwise the descriptor for the executing process is moved to the semaphore's queue. For a **V** operation, if the semaphore's queue is not empty, one descriptor is moved from that queue to the ready list; otherwise the semaphore is incremented.

This approach to implementing synchronization mechanisms is quite general and is applicable to the other mechanisms that we shall discuss. Since the kernel is responsible for allocating processor cycles to processes, it can implement a synchronization mechanism without using busy-waiting. It does this by not running processes that are blocked. Of course, the names and details of the kernel calls will differ for each synchronization mechanism, but the net effects of these calls will be similar: to move processes on and off a ready list.

Things are somewhat more complex when writing a kernel for a multiprocessor or distributed system. In a multiprocessor, either a single processor is responsible for maintaining the ready list and assigning processes to the other processors, or the ready list is shared [Jones and Schwarz, 1980]. If the ready list is shared, it is subject to concurrent access, which requires that mutual exclusion be ensured. Usually, busy-waiting is used to ensure this mutual exclusion because operations on the ready list are fast and a processor cannot execute any process until it is able to access the ready list. In a distributed system, although one processor could maintain the ready list, it is more common for each processor to have its own kernel and hence its own ready list. Each kernel manages those processes residing at one processor; if a process migrates from one processor to another, it comes under the control of the other's kernel.

3.3 Conditional Critical Regions

Although semaphores can be used to program almost any kind of synchronization, **P** and **V** are rather unstructured primitives, and so it is easy to err when using them. Execution of each critical section must begin with a **P** and end with a **V** (on the same semaphore). Omitting a **P** or **V**, or accidentally coding a **P** on one semaphore and a **V** on another can have disastrous effects, since mutually exclusive execution would no longer be ensured. Also, when using semaphores, a programmer can forget to include in critical sections all statements that reference shared objects. This, too, could destroy the mutual exclusion required within critical sections. A second difficulty with using semaphores is that both condition synchronization and mutual exclusion are programmed using the same pair of primitives. This makes it difficult to identify the purpose of a given **P** or **V** operation without looking at the other operations on the corresponding semaphore. Since mutual exclusion and condition synchronization are distinct concepts, they should have distinct notations.

The *conditional critical region* proposal [Hoare, 1972; Brinch Hansen 1972, 1973b] overcomes these difficulties by providing a structured notation for specifying synchronization. Shared variables are explicitly placed into groups, called *resources*. Each shared variable may be in at most one resource and may be accessed only in conditional critical region (CCR) statements that name the resource. Mutual exclusion is provided by guaranteeing that execution of different CCR statements, each naming the same resource, is not overlapped. Condition synchronization is provided by explicit Boolean conditions in CCR statements.

A resource r containing variables $v1$, $v2, \ldots, vN$ is declared as[13]

resource $r: v1, v2, \ldots, vN$

The variables in r may only be accessed within CCR statements that name r. Such statements have the form

region r **when** B **do** S

where B is a Boolean expression and S is a statement list. (Variables local to the executing process may also appear in the CCR statement.) A CCR statement delays the executing process until B is true; S is then executed. The evaluation of B and execution of S are uninterruptible by other CCR statements that name the same resource.

[13] Our notation combines aspects of those proposed by Hoare [1972] and by Brinch Hansen [1972, 1973b].

Thus B is guaranteed to be true when execution of S begins. The delay mechanism is usually assumed to be fair: a process awaiting a condition B that is repeatedly true will eventually be allowed to continue.

One use of conditional critical regions is shown in Figure 4, which contains another implementation of our batch operating system example. Note how condition synchronization has been separated from mutual exclusion. The Boolean expressions in those CCR statements that access the buffers explicitly specify the conditions required for access; thus mutual exclusion of different CCR statements that access the same buffer is implicit.

Programs written in terms of conditional critical regions can be understood quite simply by using the axiomatic approach. Each CCR statement implements an operation on the resource that it names. Associated with each resource r is an *invariant relation* I_r: a predicate that is true of the resource's state after the resource is initialized and after execution of any operation on the resource. For example, in *OPSYS* of Figure 4, the operations insert and remove items from bounded buffers and the buffers *inp__buff* and *out__buff* both satisfy the invariant

```
IB:
  0 ≤ head, tail ≤ N − 1 and
  0 ≤ size ≤ N and
  tail = (head + size) mod N and
  slots[head] through slots[(tail − 1) mod N]
     in the circular buffer contain
     the most recently inserted items
     in chronological order
```

The Boolean expression B in each CCR statement is chosen so that execution of the statement list, when started in any state that satisfies I_r **and** B, will terminate in a state that satisfies I_r. Therefore the invariant is true as long as no process is in the midst of executing an operation (i.e., executing in a conditional critical region associated with the resource). Recall that execution of conditional critical regions associated with a given shared data object does not overlap. Hence the proofs of processes are interference free as long as (1) variables local to a process appear only in the proof of that process and (2) variables of a resource appear only in assertions within conditional critical regions for that resource. Thus, once appropriate resource invariants have been defined, a concurrent program

```
program OPSYS;
  type buffer(T) = record
            slots : array [0..N−1] of T;
            head, tail : 0..N−1 initial (0, 0);
            size : 0..N initial (0)
            end;
  var inp_buff : buffer(cardimage);
     out_buff : buffer(lineimage);

  resource ib : inp_buff; ob : out_buff;

  process reader;
    var card : cardimage;
    loop
      read card from cardreader;
      region ib when inp_buff.size < N do
        inp_buff.slots[inp_buff.tail] := card;
        inp_buff.size := inp_buff.size + 1;
        inp_buff.tail := (inp_buff.tail + 1) mod N
        end
      end
    end;

  process executer;
    var card : cardimage;
       line : lineimage;
    loop
      region ib when inp_buff.size > 0 do
        card := inp_buff.slots[inp_buff.head]
        inp_buff.size := inp_buff.size − 1;
        inp_buff.head := (inp_buff.head + 1) mod N
        end;
      process card and generate line;
      region ob when out_buff.size < N do
        out_buff.slots[out_buff.tail] := line;
        out_buff.size := out_buff.size + 1;
        out_buff.tail := (out_buff.tail + 1) mod N
        end
      end
    end;

  process printer;
    var line : lineimage;
    loop
      region ob when out_buff.size > 0 do
        line := out_buff.slots[out_buff.head];
        out_buff.size := out_buff.size − 1;
        out_buff.head := (out_buff.head + 1) mod N
        end;
      print line on lineprinter
      end
    end

  end.
```

Figure 4. Batch operating system with CCR statements.

can be understood in terms of its component sequential processes.

Although conditional critical regions have many virtues, they can be expensive to implement. Because conditions in CCR statements can contain references to local variables, each process must evaluate its own conditions.[14] On a multiprogrammed processor, this evaluation results in numerous context switches (frequent saving and restoring of process states), many of which may be unproductive because the activated process may still find the condition false. If each process is executed on its own processor and memory is shared, however, CCR statements can be implemented quite cheaply by using busy-waiting.

CCR statements provide the synchronization mechanism in the Edison language [Brinch Hansen, 1981], which is designed specifically for multiprocessor systems. Variants have also been used in Distributed Processes [Brinch Hansen, 1978] and Argus [Liskov and Scheifler, 1982].

3.4 Monitors

Conditional critical regions are costly to implement on single processors. Also, CCR statements performing operations on resource variables are dispersed throughout the processes. This means that one has to study an entire concurrent program to see all the ways in which a resource is used. Monitors alleviate both these deficiencies. A *monitor* is formed by encapsulating both a resource definition and operations that manipulate it [Dijkstra, 1968b; Brinch Hansen, 1973a; Hoare, 1974]. This allows a resource subject to concurrent access to be viewed as a module [Parnas, 1972]. Consequently, a programmer can ignore the implementation details of the resource when using it, and can ignore how it is used when programming the monitor that implements it.

3.4.1 Definition

A monitor consists of a collection of *permanent variables*, used to store the resource's state, and some procedures, which implement operations on the resource. A monitor also has permanent-variable initialization code, which is executed once before any procedure body is executed. The values of the permanent variables are retained between activations of monitor procedures and may be accessed only from within the monitor. Monitor procedures can have parameters and local variables, each of which takes on new values for each procedure activation. The structure of a monitor with name *mname* and procedures *op1*, . . . , *opN* is shown in Figure 5.

```
mname : monitor;
  var declarations of permanent variables;

  procedure op1(parameters);
    var declarations of variables local to op1;
    begin
      code to implement op1
    end;

  ...

  procedure opN(parameters);
    var declarations of variables local to opN;
    begin
      code to implement opN
    end;

  begin
    code to initialize permanent variables
  end
```

Figure 5. Monitor structure.

Procedure *opJ* within monitor *mname* is invoked by executing

call *mname.opJ*(arguments).

The invocation has the usual semantics associated with a procedure call. In addition, execution of the procedures in a given monitor is guaranteed to be mutually exclusive. This ensures that the permanent variables are never accessed concurrently.

A variety of constructs have been proposed for realizing condition synchronization in monitors. We first describe the proposal made by Hoare [1974] and then consider other proposals. A *condition variable* is used to delay processes executing in a monitor; it may be declared only within a monitor. Two operations are defined on condition variables: **signal** and **wait**. If

[14] When delayed, a process could instead place condition evaluating code in an area of memory accessible to other processes, but this too is costly.

```
type buffer(T) = monitor;
  var { the variables satisfy invariant IB — see Sec. 4.3 }
      slots : array [0..N−1] of T;
      head, tail : 0..N−1;
      size : 0..N;
      notfull, notempty : condition;
  procedure deposit(p : T);
    begin
      if size = N  then notfull.wait;
      slots[tail] := p;
      size := size + 1;
      tail := (tail + 1) mod N;
      notempty.signal
    end;
  procedure fetch(var it : T);
    begin
      if size = 0  then notempty.wait;
      it := slots[head];
      size := size − 1;
      head := (head + 1) mod N;
      notfull.signal
    end;
  begin
    size := 0;  head := 0;  tail := 0
  end
```

Figure 6. Bounded buffer monitor.

```
program OPSYS;
  type buffer(T) = ...;  { see Figure 5 }
  var inp_buff : buffer(cardimage);
      out_buff : buffer(lineimage);
  process reader;
    var card : cardimage;
    loop
      read card from cardreader;
      call inp_buff.deposit(card)
      end
    end;
  process executer;
    var card : cardimage;
        line : lineimage;
    loop
      call inp_buff.fetch(card);
      process card and generate line;
      call out_buff.deposit(line)
      end
    end;
  process printer;
    var line : lineimage;
    loop
      call out_buff.fetch(line);
      print line on lineprinter
      end
    end
  end.
```

Figure 7. Batch operating system with monitors.

cond is a condition variable, then execution of

cond.**wait**

causes the invoker to be blocked on *cond* and to relinquish its mutually exclusive control of the monitor. Execution of

cond.**signal**

works as follows: if no process is blocked on *cond*, the invoker continues; otherwise, the invoker is temporarily suspended and one process blocked on *cond* is reactivated. A process suspended due to a **signal** operation continues when there is no other process executing in the monitor. Moreover, signalers are given priority over processes trying to commence execution of a monitor procedure. Condition variables are assumed to be fair in the sense that a process will not forever remain suspended on a condition variable that is signaled infinitely often. Note that the introduction of condition variables allows more than one process to be in the same monitor, although all but one will be delayed at **wait** or **signal** operations.

An example of a monitor that defines a bounded buffer type is given in Figure 6. Our batch operating system can be programmed using two instances of the bounded buffer in Figure 6; these are shared by three processes, as shown in Figure 7.

At times, a programmer requires more control over the order in which delayed processes are awakened. To implement such *medium-term scheduling*,[15] the *priority* **wait** statement can be used. This statement

cond.**wait**(p)

has the same semantics as *cond*.**wait**, except that in the former processes blocked on condition variable *cond* are awakened in ascending order of p. (Consequently, con-

[15] This is in contrast to *short-term scheduling*, which is concerned with how processors are assigned to ready processes, and *long-term scheduling*, which refers to how jobs are selected to be processed.

dition variables used in this way are not necessarily fair.)

A common problem involving medium-term scheduling is "shortest-job-next" resource allocation. A resource is to be allocated to at most one user at a time; if more than one user is waiting for the resource when it is released, it is allocated to the user who will use it for the shortest amount of time. A monitor to implement such an allocator is shown below. The monitor has two procedures: (1) *request*(*time*: *integer*), which is called by users to request access to the resource for *time* units; and (2) *release*, which is called by users to relinquish access to the resource:

```
shortest_next_allocator : monitor;

  var free : Boolean;
      turn : condition;

  procedure request(time : integer);
    begin
      if not free then  turn.wait(time);
      free := false
    end;

  procedure release;
    begin
      free := true;
      turn.signal
    end;

  begin
    free := true
  end
```

3.4.2 Other Approaches to Condition Synchronization

3.4.2.1 Queues and Delay/Continue. In Concurrent PASCAL [Brinch Hansen, 1975], a slightly simpler mechanism is provided for implementing condition synchronization and medium-term scheduling. Variables of type *queue* can be defined and manipulated with the operations **delay** (analogous to **wait**) and **continue** (analogous to **signal**). In contrast to condition variables, at most one process can be suspended on a given *queue* at any time. This allows medium-term scheduling to be implemented by (1) defining an array of queues and (2) performing a **continue** operation on that queue on which the next-process-to-be-awakened has been delayed. The semantics of **continue** are also slightly different from **signal**. Executing **continue** causes the invoker to return from its monitor call, whereas **signal** does not. As before, a process blocked on the selected queue resumes execution of the monitor procedure within which it was delayed.

It is both cheaper and easier to implement **continue** than **signal** because **signal** requires code to ensure that processes suspended by **signal** operations reacquire control of the monitor before other, newer processes attempting to begin execution in that monitor. With both **signal** and **continue**, the objective is to ensure that a condition is not invalidated between the time it is signaled and the time that the awakened process actually resumes execution. Although **continue** has speed and cost advantages, it is less powerful than **signal**. A monitor written using condition variables cannot always be translated directly into one that uses queues without also adding monitor procedures [Howard, 1976b]. Clearly, these additional procedures complicate the interface provided by the monitor. Fortunately, most synchronization problems that arise in practice can be coded using either discipline.

3.4.2.2 Conditional Wait and Automatic Signal. In contrast to semaphores, **signals** on condition variables are not saved: a process always delays after executing **wait**, even if a previous **signal** did not awaken any process.[16] This can make **signal** and **wait** difficult to use correctly, because other variables must be used to record that a **signal** was executed. These variables must also be tested by a process, before executing **wait**, to guard against waiting if the event corresponding to a **signal** has already occurred.

Another difficulty is that, in contrast to conditional critical regions, a Boolean expression is not syntactically associated with **signal** and **wait**, or with the condition variable itself. Thus, it is not easy to determine why a process was delayed on a condition variable, unless **signal** and **wait** are used in a very disciplined manner. It helps if (1) each **wait** on a condition variable is

[16] The limitations of condition variables discussed in this section also apply to queue variables.

contained in an **if** statement in which the Boolean expression is the negation of the desired condition synchronization, and (2) each **signal** statement on the same condition variable is contained in an **if** statement in which the Boolean expression gives the desired condition synchronization. Even so, syntactically identical Boolean expressions may have different values if they contain references to local variables, which they often do. Thus there is no guarantee that an awakened process will actually see the condition for which it was waiting. A final difficulty with **wait** and **signal** is that, because **signal** is preemptive, the state of permanent variables seen by a signaler can change between the time a **signal** is executed and the time that the signaling process resumes execution.

To mitigate these difficulties, Hoare [1974] proposed the *conditional wait* statement

wait (B)

where B is a Boolean expression involving the permanent or local variables of the monitor. Execution of **wait**(B) delays the invoker until B becomes true; no **signal** is required to reactivate processes delayed by a conditional wait statement. This synchronization facility is expensive because it is necessary to evaluate B every time any process exits the monitor or becomes blocked at a conditional wait and because a context switch could be required for each evaluation (due to the presence of local variables in the condition). However, the construct is unquestionably a very clean one with which to program.

An efficient variant of the conditonal wait was proposed by Kessels [1977] for use when only permanent variables appear in B. The buffer monitor in Figure 6 satisfies this requirement. In Kessels' proposal, one declares *conditions* of the form

cname : **condition** B

Executing the statement *cname*.**wait** causes B, a Boolean expression, to be evaluated. If B is true, the process continues; otherwise the process relinquishes control of the monitor and is delayed on *cname*. Whenever a process relinquishes control of the monitor, the system evaluates those Boolean expressions associated with all conditions for which there are waiting processes. If one of these Boolean expressions is found to be true, one of the waiting processes is granted control of the monitor. If none is found to be true, a new invocation of one of the monitor's procedures is permitted.

Using Kessels' proposal, the buffer monitor in Figure 6 could be recoded as follows. First, the declarations of *not_full* and *not_empty* are changed to

not_full : **condition** $size < N$;
not_empty : **condition** $size > 0$

Second, the first statement in *deposit* is replaced by

not_full.**wait**

and the first statement in *fetch* is replaced by

not_empty.**wait**

Finally, the **signal** statements are deleted.

The absence of a **signal** primitive is noteworthy. The implementation provides an *automatic signal*, which, though somewhat more costly, is less error prone than explicitly programmed **signal** operations. The **signal** operation cannot be accidentally omitted and never signals the wrong condition. Furthermore, the programmer explicitly specifies the conditions being awaited. The primary limitation of the proposal is that it cannot be used to solve most scheduling problems, because operation parameters, which are not permanent variables, may not appear in conditions.

3.4.2.3 Signals as Hints. Mesa [Mitchell et al., 1979; Lampson and Redell, 1980] employs yet another approach to condition synchronization. Condition variables are provided, but only as a way for a process to relinquish control of a monitor. In Mesa, execution of

cond.**notify**

causes a process waiting on condition variable *cond* to resume at some time in the future. This is called *signal and continue* because the process performing the **notify** immediately continues execution rather than being suspended. Performing a **notify** merely gives a *hint* to a waiting process

that it might be able to proceed.[17] Therefore, in Mesa one writes

while not B **do wait** *cond* **endloop**

instead of

if not B **then** *cond*.**wait**

as would be done using Hoare's condition variables. Boolean condition B is guaranteed to be true upon termination of the loop, as it was in the two conditional-wait/automatic-signal proposals. Moreover, the (possible) repeated evaluation of the Boolean expression appears in the actual monitor code—there are no hidden implementation costs.

The **notify** primitive is especially useful if the executing process has higher priority than the waiting processes. It also allows the following extensions to condition variables, which are often useful when doing systems programming:

(i) A time-out interval t can be associated with each condition variable. If a process is ever suspended on this condition variable for longer than t time units, a **notify** is automatically performed by the system. The awakened process can then decide whether to perform another **wait** or to take other action.

(ii) A **broadcast** primitive can be defined. Its execution causes all processes waiting on a condition variable to resume at some time in the future (subject to the mutual exclusion constraints associated with execution in a monitor). This primitive is useful if more than one process could proceed when a condition becomes true. The broadcast primitive is also useful when a condition involves local variables because in this case the signaler cannot evaluate the condition (B above) for which a process is waiting. Such a primitive is, in fact, used in UNIX [Ritchie and Thompson, 1974].

3.4.3 An Axiomatic View

The valid states of a resource protected by a monitor can be characterized by an assertion called the *monitor invariant*. This predicate should be true of the monitor's permanent variables whenever no process is executing in the monitor. Thus a process must reestablish the monitor invariant before the process exits the monitor or performs a **wait(delay)** or **signal(continue)**. The monitor invariant can be assumed to be true of the permanent variables whenever a process acquires control of the monitor, regardless of whether it acquires control by calling a monitor procedure or by being reactivated following a **wait** or **signal**.

The fact that monitor procedures are mutually exclusive simplifies noninterference proofs. One need not consider interleaved execution of monitor procedures. However, interference can arise when programming condition synchronization. Recall that a process will delay its progress in order to implement medium-term scheduling, or to await some condition. Mechanisms that delay a process cause its execution to be suspended and control of the monitor to be relinquished; the process resumes execution with the understanding that both some condition B and the monitor invariant will be true. The truth of B when the process awakens can be ensured by checking for it automatically or by requiring that the programmer build these tests into the program. If programmed checks are used, they can appear either in the process that establishes the condition (for condition variables and queues) or in the process that performed the **wait** (the Mesa model).

If the signaler checks for the condition, we must ensure that the condition is not invalidated between the time that the **signal** occurs and the time that the blocked process actually executes. That is, we must ensure that other execution in the monitor does not interfere with the condition. If the signaler does not immediately relinquish control of the monitor (e.g., if **notify** is used), interference might be caused by the process that established the condition in the first place. Also, if the signaled process does not get reactivated before new calls of monitor procedures are allowed, interference might be caused by some process that executes after the condition has been signaled (this can happen in Modula [Wirth,

[17] Of course, it is prudent to perform **notify** operations only when there is reason to believe that the awakened process will actually be able to proceed; but the burden of checking the condition is on the waiting process.

1977a]). Proof rules for monitors and the various signaling disciplines are discussed by Howard [1976a, 1976b].

3.4.4 Nested Monitor Calls

When structuring a system as a hierarchical collection of monitors, it is likely that monitor procedures will be called from within other monitors. Such nested monitor calls have caused much discussion [Haddon, 1977; Lister, 1977; Parnas, 1978; Wettstein, 1978]. The controversy is over what (if anything) should be done if a process having made a nested monitor call is suspended in another monitor. The mutual exclusion in the last monitor called will be relinquished by the process, due to the semantics of **wait** and equivalent operations. However, mutual exclusion will not be relinquished by processes in monitors from which nested calls have been made. Processes that attempt to invoke procedures in these monitors will become blocked. This has performance implications, since blockage will decrease the amount of concurrency exhibited by the system.

The nested monitor call problem can be approached in a number of ways. One approach is to prohibit nested monitor calls, as was done in SIMONE [Kaubisch et al., 1976], or or to prohibit nested calls to monitors that are not lexically nested, as was done in Modula [Wirth, 1977a]. A second approach is to release the mutual exclusion on all monitors along the call chain when a nested call is made and that process becomes blocked.[18] This release-and-reacquire approach would require that the monitor invariant be established before any monitor call that will block the process. Since the designer cannot know a priori whether a call will block a process, the monitor invariant would have to be established before every call. A third approach is the definition of special-purpose constructs that can be used for particular situations in which nested calls often arise. The *manager* construct [Silberschatz et al., 1977] for handling dynamic resource allocation problems and the *scheduler monitor* [Schneider and Bernstein, 1978] for scheduling access to shared resources are both based on this line of thought.

The last approach to the nested monitor call problem, and probably the most reasonable, is one that appreciates that monitors are only a structuring tool for resources that are subject to concurrent access [Andrews and McGraw, 1977; Parnas, 1978]. Mutual exclusion of monitor procedures is only one way to preserve the integrity of the permanent variables that make up a resource. There are cases in which the operations provided by a given monitor can be executed concurrently without adverse effects, and even cases in which more than one instance of the same monitor procedure can be executed in parallel (e.g., several activations of a read procedure, in a monitor that encapsulates a database). Monitor procedures can be executed concurrently, provided that they do not interfere with each other. Also, there are cases in which the monitor invariant can be easily established before a nested monitor call is made, and so mutual exclusion for the monitor can be released. Based on such reasoning, Andrews and McGraw [1977] defines a monitorlike construct that allows the programmer to specify that certain monitor procedures be executed concurrently and that mutual exclusion be released for certain calls. The Mesa language [Mitchell et al., 1979] also provides mechanisms that give the programmer control over the granularity of exclusion.

3.4.5 Programming Notations Based on Monitors

Numerous programming languages have been proposed and implemented that use monitors for synchronizing access to shared variables. Below, we very briefly discuss two of the most important: Concurrent PASCAL and Modula. These languages have received widespread use, introduced novel constructs to handle machine-dependent systems-programming issues, and inspired other language designs, such as Mesa [Mitchell et al., 1979] and PASCAL-Plus [Welsh and Bustard, 1979].

[18] Once signaled, the process will need to reacquire exclusive access to all monitors along the call chain before resuming execution. However, if permanent monitor variables were not passed as reference parameters in any of the calls, the process could reacquire exclusive access incrementally, as it returns to each monitor.

3.4.5.1 Concurrent PASCAL. Concurrent PASCAL [Brinch Hansen, 1975, 1977] was the first programming language to support monitors. Consequently, it provided a vehicle for evaluating monitors as a system-structuring device. The language has been used to write several operating systems, including Solo, a single-user operating system [Brinch Hansen, 1976a, 1976b], Job Stream, a batch operating system for processing PASCAL programs, and a real-time process control system [Brinch Hansen, 1977].

One of the major goals of Concurrent PASCAL was to ensure that programs exhibited reproducible behavior [Brinch Hansen, 1977]. Monitors ensured that pathological interleavings of concurrently executed routines that shared data were no longer possible (the compiler generates code to provide the necessary mutual exclusion). Concurrent execution in other modules (called *classes*) was not possible, due to compile-time restrictions on the dissemination of class names and scope rules for class declarations.

Concurrent PASCAL also succeeded in providing the programmer with a clean abstract machine, thereby eliminating the need for coding at the assembly language level. A systems programming language must have facilities to allow access to I/O devices and other hardware resources. In Concurrent PASCAL, I/O devices and the like are viewed as monitors implemented directly in hardware. To perform an I/O operation, the corresponding "monitor" is called; the call returns when the I/O has completed. Thus the Concurrent PASCAL run-time system implements synchronous I/O and "abstracts out" the notion of an interrupt.

Various aspects of Concurrent PASCAL, including its approach to I/O, have been analyzed by Loehr [1977], Silberschatz [1977], and Keedy [1979].

3.4.5.2 Modula. Modula was developed for programming small, dedicated computer systems, including process control applications [Wirth, 1977a, 1977b, 1977c, 1977d]. The language is largely based on PASCAL and includes processes, *interface modules*, which are like monitors, and *device modules*, which are special interface modules for programming device drivers.

The run-time support system for Modula is small and efficient. The kernel for a PDP-11/45 requires only 98 words of storage and is extremely fast [Wirth, 1977c]. It does not time slice the processor among processes, as Concurrent PASCAL does. Rather, certain kernel-supported operations—**wait**, for example—always cause the processor to be switched. (The programmer must be aware of this and design programs accordingly.) This turns out to be both a strength and weakness of Modula. A small and efficient kernel, where the programmer has some control over processor switching, allows Modula to be used for process control applications, as intended. Unfortunately, in order to be able to construct such a kernel, some of the constructs in the language—notably those concerning multiprogramming—have associated restrictions that can only be understood in terms of the kernel's implementation. A variety of subtle interactions between the various synchronization constructs must be understood in order to program in Modula without experiencing unpleasant surprises. Some of these pathological interactions are described by Bernstein and Ensor [1981].

Modula implements an abstract machine that is well suited for dealing with interrupts and I/O devices on PDP-11 processors. Unlike Concurrent PASCAL, in which the run-time kernel handles interrupts and I/O, Modula leaves support for devices in the programmer's domain. Thus new devices can be added without modifying the kernel. An I/O device is considered to be a process that is implemented in hardware. A software process can start an I/O operation and then execute a **doio** statement (which is like a **wait** except that it delays the invoker until the kernel receives an interrupt from the corresponding device). Thus interrupts are viewed as **signal** (**send** in Modula) operations generated by the hardware. Device modules are interface modules that control I/O devices. Each contains, in addition to some procedures, a *device process*, which starts I/O operations and executes **doio** statements to relinquish control of the processor (pending receipt of the corresponding I/O interrupt). The address

of the interrupt vector for the device is declared in the heading of the device module, so that the compiler can do the necessary binding. Modula also has provisions for controlling the processor priority register, thus allowing a programmer to exploit the priority interrupt architecture of the processor when structuring programs.

A third novel aspect of Modula is that variables declared in interface modules can be exported. Exported variables can be referenced (but not modified) from outside the scope of their defining interface module. This allows concurrent access to these variables, which, of course, can lead to difficulty unless the programmer ensures that interference cannot occur. However, when used selectively, this feature increases the efficiency of programs that access such variables.

In summary, Modula is less constraining than Concurrent PASCAL, but requires the programmer to be more careful. Its specific strengths and weaknesses have been evaluated by Andrews [1979], Holden and Wand [1980], and Bernstein and Ensor [1981]. Wirth, Modula's designer, has gone on to develop Modula-2 [Wirth, 1982]. Modula-2 retains the basic modular structure of Modula, but provides more flexible facilities for concurrent programming and these facilities have less subtle semantics. In particular, Modula-2 provides coroutines and hence explicit transfer of control between processes. Using these, the programmer builds support for exclusion and condition synchronization, as required. In particular, the programmer can construct monitorlike modules.

3.5 Path Expressions

Operations defined by a monitor are executed with mutual exclusion. Other synchronization of monitor procedures is realized by explicitly performing **wait** and **signal** operations on condition variables (or by some similar mechanism). Consequently, synchronization of monitor operations is realized by code scattered throughout the monitor. Some of this code, such as **wait** and **signal**, is visible to the programmer. Other code, such as the code ensuring mutual exclusion of monitor procedures, is not.

Another approach to defining a module subject to concurrent access is to provide a mechanism with which a programmer specifies, in *one* place in each module, all constraints on the execution of operations defined by that module. Implementation of the operations is separated from the specification of the constraints. Moreover, code to enforce the constraints is generated by a compiler. This is the approach taken in a class of synchronization mechanisms called *path expressions.*

Path expressions were first defined by Campbell and Habermann [1974]. Subsequent extensions and variations have also been proposed [Habermann, 1975; Lauer and Campbell, 1975; Campbell, 1976; Flon and Habermann, 1976; Lauer and Shields, 1978; Andler, 1979]. Below, we describe one specific proposal [Campbell, 1976] that has been incorporated into Path PASCAL, an implemented systems programming language [Campbell and Kolstad, 1979].

When path expressions are used, a module that implements a resource has a structure like that of a monitor. It contains permanent variables, which store the state of the resource, and procedures, which realize operations on the resource. Path expressions in the header of each resource define constraints on the order in which operations are executed. No synchronization code is programmed in the procedures.

The syntax of a path expression is

path *path_list* **end**

A *path_list* contains operation names and *path operators.* Path operators include "," for concurrency, ";" for sequencing, "*n* : (*path_list*)" to specify up to *n* concurrent activations of *path_list*, and "[*path_list*]" to specify an unbounded number of concurrent activations of *path_list.*

For example, the path expression

path *deposit, fetch* **end**

places no constraints on the order of execution of *deposit* and *fetch* and no constraints on the number of activations of either operation. This absence of synchronization constraints is equivalent to that

specified by the path expressions

path [*deposit*], [*fetch*] **end**

or

path [*deposit, fetch*] **end**

(A useful application of the "[· · ·]" operator will be shown later.) In contrast,

path *deposit*; *fetch* **end**

specifies that each *fetch* be preceded by a *deposit*; multiple activations of each operation can execute concurrently as long as the number of active or completed *fetch* operations never exceeds the number of completed *deposit* operations. A module implementing a bounded buffer of size one might well contain the path

path 1 : (*deposit*; *fetch*) **end**

to specify that the first invoked operation be a *deposit*, that each *deposit* be followed by a *fetch*, and that at most one instance of the path "*deposit*; *fetch*" be active—in short, that *deposit* and *fetch* alternate and are mutually exclusive. Synchronization constraints for a bounded buffer of size *N* are specified by

path *N* : (1 : (*deposit*); 1 : (*fetch*)) **end**

This ensures that (i) activations of *deposit* are mutually exclusive, (ii) activations of *fetch* are mutually exclusive, (iii) each activation of *fetch* is preceded by a completed *deposit*, and (iv) the number of completed *deposit* operations is never more than *N* greater than the number of completed *fetch* operations. The bounded buffers we have been using for *OPSYS*, our batch operating system, would be defined by

```
module buffer(T);
  path N:( 1:(deposit); 1:(fetch) ) end;
  var { the variables satisfy the invariant IB (see Sec. 4.3)
        with size equal to the number of executions of
        deposit minus the number of executions of fetch }
     slots : array [0..N-1] of T;
     head, tail : 0..N-1;
  procedure deposit(p : T);
    begin
      slots[tail] := p;
      tail := (tail + 1) mod N
    end;
  procedure fetch(var it : T);
    begin
      it := slots[head];
      head := (head + 1) mod N
    end;
  begin
    head := 0; tail := 0
  end.
```

Note that one *deposit* and one *fetch* can proceed concurrently, which was not possible in the *buffer* monitor given in Figure 6. For this reason, there is no variable *size* because it would have been subject to concurrent access.

As a last example, consider the readers/writers problem [Courtois et al., 1971]. In this problem, processes read or write records in a shared data base. To ensure that processes read consistent data, either an unbounded number of concurrent *reads* or a single *write* may be executed at any time. The path expression

path 1 : ([*read*], *write*) **end**

specifies this constraint. (Actually, this specifies the "weak reader's preference" solution to the readers/writers problem: readers can prevent writers from accessing the database.)

Path expressions are strongly motivated by, and based on, the operational approach to program semantics. A path expression defines all legal sequences of the operation executions for a resource. This set of sequences can be viewed as a formal language, in which each sentence is a sequence of operation names. In light of this, the resemblance between path expressions and regular expressions should not be surprising.

While path expressions provide an elegant notation for expressing synchronization constraints described operationally, they are poorly suited for specifying condition synchronization [Bloom, 1979]. Whether an operation can be executed might depend on the state of a resource in a way not directly related to the history of operations already performed. Certain variants of the readers/writers problem (e.g., writers preference, fair access for readers and writers) require access to the state of the resource—in this case, the number of waiting readers and waiting writers—in order to implement the desired synchronization. The *shortest_next_allocator* monitor of Section 3.4.1 is an example of a resource in which a parameter's value determines whether execution of an operation (*request*) should be permitted to continue.

In fact, most resources that involve scheduling require access to parameters and/or to state information when making synchronization decisions. In order to use path expressions to specify solutions to such problems, additional mechanisms must be introduced. In some cases, definition of additional operations on the resource is sufficient; in other cases "queue" resources, which allow a process to suspend itself and be reactivated by a "scheduler," must be added. The desire to realize condition synchronization using path expressions has motivated many of the proposed extensions. Regrettably, none of these extensions have solved the entire problem in a way consistent with the elegance and simplicity of the original proposal. However, path expressions have proved useful for specifying the semantics of concurrent computations [Shields, 1979; Shaw, 1980, Best, 1982].

4. SYNCHRONIZATION PRIMITIVES BASED ON MESSAGE PASSING

Critical regions, monitors, and path expressions are one outgrowth of semaphores; they all provide structured ways to control access to shared variables. A different outgrowth is *message passing*, which can be viewed as extending semaphores to convey data as well as to implement synchronization. When message passing is used for communication and synchronization, processes send and receive messages instead of reading and writing shared variables. Communication is accomplished because a process, upon receiving a message, obtains values from some sender process. Synchronization is accomplished because a message can be received only after it has been sent, which constrains the order in which these two events can occur.

A message is sent by executing

send *expression_list*
  to *destination_designator.*

The message contains the values of the expressions in *expression_list* at the time **send** is executed. The *destination_designator* gives the programmer control over where the message goes, and hence over which statements can receive it. A message is received by executing

receive *variable_list*
  from *source_designator*

where *variable_list* is a list of variables. The *source_designator* gives the programmer control over where the message came from, and hence over which statements could have sent it. Receipt of a message causes, first, assignment of the values in the message to the variables in *variable_list* and, second, subsequent destruction of the message.[19]

Designing message-passing primitives involves making choices about the form and semantics of these general commands. Two main issues must be addressed: How are source and destination designators specified? How is communication synchronized? Common alternative solutions for these issues are described in the next two sections. Then higher level message-passing constructs, semantic issues, and languages based on message passing are discussed.

4.1 Specifying Channels of Communication

Taken together, the destination and source designators define a *communications channel.* Various schemes have been proposed for naming channels. The simplest channel-naming scheme is for process names to serve as source and destination designators. We refer to this as *direct naming*. Thus

send *card* **to** *executer*

sends a message that can be received only by the *executer* process. Similarly,

receive *line* **from** *executer*

permits receipt only of a message sent by the executer process.

Direct naming is easy to implement and to use. It makes it possible for a process to control the times at which it receives messages from each other process. Our simple batch operating system might be programmed using direct naming as shown in Figure 8.

The batch operating system also illustrates an important paradigm for process

[19] A broadcast can be modeled by the concurrent execution of a collection of **sends**, each sending the message to a different destination. A nondestructive receive can be modeled by a **receive**, immediately followed by a **send**.

```
program OPSYS;
  process reader;
    var card : cardimage;
    loop
      read card from cardreader;
      send card to executer
      end
    end;
  process executer;
    var card : cardimage; line : lineimage;
    loop
      receive card from reader;
      process card and generate line;
      send line to printer
      end
    end;
  process printer;
    var line : lineimage;
    loop
      receive line from executer;
      print line on lineprinter
      end
    end
  end.
```

Figure 8. Batch operating system with message passing.

interaction—a pipeline. A *pipeline* is a collection of concurrent processes in which the output of each process is used as the input to another. Information flows analogously to the way liquid flows in a pipeline. Here, information flows from the *reader* process to the *executer* process and then from the *executer* process to the *printer* process. Direct naming is particularly well suited for programming pipelines.

Another important paradigm for process interaction is the *client/server relationship*. Some *server* processes render a service to some *client* processes. A client can request that a service be performed by sending a message to one of these servers. A server repeatedly receives a request for service from a client, performs that service, and (if necessary) returns a completion message to that client.

The interaction between an I/O driver process and processes that use it—for example, the lineprinter driver and the *printer* process in our operating system example—illustrates this paradigm. The lineprinter driver is a server; it repeatedly receives requests to print a line on the printer, starts that I/O operation, and then awaits the interrupt signifying completion of the I/O operation. Depending on the application, it might also send a completion message to the client after the line has been printed.

Unfortunately, direct naming is not always well suited for client/server interaction. Ideally, the **receive** in a server should allow receipt of a message from any client. If there is only one client, then direct naming will work well; difficulties arise if there is more than one client because, at the very least, a **receive** would be required for each. Similarly, if there is more than one server (and all servers are identical), then the **send** in a client should produce a message that can be received by *any* server. Again, this cannot be accomplished easily with direct naming. Therefore, a more sophisticated scheme for defining communications channels is required.

One such scheme is based on the use of *global names*, sometimes called *mailboxes*. A mailbox can appear as the destination designator in any process' **send** statements and as the source designator in any process' **receive** statements. Thus messages sent to a given mailbox can be received by any process that executes a **receive** naming that mailbox.

This scheme is particularly well suited for programming client/server interactions. Clients send their service requests to a single mailbox; servers receive service requests from that mailbox. Unfortunately, implementing mailboxes can be quite costly without a specialized communications network [Gelernter and Bernstein, 1982]. When a message is sent, it must be relayed to all sites where a **receive** could be performed on the destination mailbox; then, after a message has been received, all these sites must be notified that the message is no longer available for receipt.

The special case of mailboxes, in which a mailbox name can appear as the source designator in **receive** statements in one process only, does not suffer these implementation difficulties. Such mailboxes are often called *ports* [Balzer, 1971]. Ports are simple to implement, since all **receives** that designate a port occur in the same process. Moreover, ports allow a straightforward solution to the multiple-clients/

single-server problem. (The multiple-clients/multiple-server problem, however, is not easily solved with ports.)

To summarize, when direct naming is used, communication is one to one since each communicating process names the other. When port naming is used, communication can be many to one since each port has one receiver but may have many senders. The most general scheme is global naming, which can be many to many. Direct naming and port naming are special cases of global naming; they limit the kinds of interactions that can be programmed directly, but are more efficient to implement.

Source and destination designators can be fixed at compile time, called *static channel naming*, or they can be computed at run time, called *dynamic channel naming.* Although widely used, static naming presents two problems. First, it precludes a program from communicating along channels not known at compile time, and thus limits the program's ability to exist in a changing environment. For example, this would preclude implementing the I/O redirection or pipelines provided by UNIX [Ritchie and Thompson, 1974].[20] The second problem is this: if a program might *ever* need access to a channel, it must permanently have the access. In many applications, such as file systems, it is more desirable to allocate communications channels to resources (such as files) dynamically.

To support dynamic channel naming, an underlying, static channel-naming scheme could be augmented by variables that contain source or destination designators. These variables can be viewed as containing *capabilities* for the communications channel [Baskett et al., 1977; Solomon and Finkel, 1979; Andrews, 1982].

4.2 Synchronization

Another important property of message-passing statements concerns whether their execution could cause a delay. A statement is *nonblocking* if its execution never delays its invoker; otherwise the statement is said to be *blocking*. In some message-passing schemes, messages are buffered between the time they are sent and received. Then, if the buffer is full when a **send** is executed, there are two options: the **send** might delay until there is space in the buffer for the message, or the **send** might return a code to the invoker, indicating that, because the buffer was full, the message could not be sent. Similarly, execution of a **receive**, when no message that satisfies the source designator is available for receipt, might either cause a delay or terminate with a code, signifying that no message was available.

If the system has an effectively unbounded buffer capacity, then a process is never delayed when executing a **send**. This is variously called *asynchronous message passing* and *send no-wait*. Asynchronous message passing allows a sender to get arbitrarily far ahead of a receiver. Consequently, when a message is received, it contains information about the sender's state that is not necessarily still its current state. At the other extreme, with no buffering, execution of a **send** is always delayed until a corresponding[21] **receive** is executed; then the message is transferred and both proceed. This is called *synchronous message passing*. When synchronous message passing is used, a message exchange represents a synchronization point in the execution of both the sender and receiver. Therefore, the message received will always correspond to the sender's current state. Moreover, when the **send** terminates, the sender can make assertions about the state of the receiver. Between these two extremes is *buffered message passing*, in which the buffer has finite bounds. Buffered message passing allows the sender to get ahead of the receiver, but not arbitrarily far ahead.

The blocking form of the **receive** statement is the most common, because a receiving process often has nothing else to do while awaiting receipt of a message. However, most languages and operating systems

[20] Although in UNIX most commands read from and write to the user's terminal, one can specify that a command read its input from a file or write its output to a file. Also, one can specify that commands be connected in a pipeline. These options are provided by a dynamic channel-naming scheme that is transparent to the implementation of each command.

[21] Correspondence is determined by the source and destination designators.

also provide a nonblocking **receive** or a means to test whether execution of a **receive** would block. This enables a process to receive all available messages and then select one to process (effectively, to schedule them).

Sometimes, further control over which messages can be received is provided. The statement

```
receive variable_list
    from source_designator when B
```

permits receipt of only those messages that make *B* true. This allows a process to "peek" at the contents of a delivered message before receiving it. Although this facility is not necessary—a process can always receive and store copies of messages until appropriate to act on them, as shown in the shortest-next-allocator example at the end of this section—the conditional receive makes possible concise solutions to many synchronization problems. Two languages that provide such a facility, PLITS and SR, are described in Section 4.5.

A blocking **receive** implicitly implements synchronization between sender and receiver because the receiver is delayed until after the message is sent. To implement such synchronization with nonblocking **receive**, busy-waiting is required. However, blocking message-passing statements can achieve the same semantic effects as nonblocking ones by using what we shall call *selective communications*, which is based on Dijkstra's guarded commands [Dijkstra, 1975].

In a selective-communications statement, a *guarded command* has the form

guard → *statement*

The guard consists of a Boolean expression, optionally followed by a message-passing statement. The guard *succeeds* if the Boolean expression is true and executing the message-passing statement would not cause a delay; the guard *fails* if the Boolean expression is false; the guard (temporarily) neither succeeds nor fails if the Boolean expression is true but the message-passing statement cannot yet be executed without causing delay. The alternative statement

```
if G1 → S1
[] G2 → S2
   ...
[] Gn → Sn
fi
```

is executed as follows. If at least one guard succeeds, one of them, *Gi*, is selected nondeterministically; the message-passing statement in *Gi* is executed (if present); then *Si*, the statement following the guard, is executed. If all guards fail, the command aborts. If all guards neither succeed nor fail, execution is delayed until some guard succeeds. (Obviously, deadlock could result.) Execution of the iterative statement is the same as for the alternative statement, except selection and execution of a guarded command is repeated until all guards fail, at which time the iterative statement terminates rather than aborts.

To illustrate the use of selective communications, we implement a *buffer* process, which stores data produced by a *producer* process and allows these data to be retrieved by a *consumer* process:[22]

```
process buffer;
  var slots : array [0..N−1] of T;
      head, tail : 0..N−1;
      size : 0..N;

  head := 0; tail := 0; size := 0;
  do size<N; receive slots[tail] from producer →
         size := size + 1;
         tail := (tail + 1) mod N
  [] size>0; send slots[head] to consumer →
         size := size − 1;
         head := (head + 1) mod N
  od
  end
```

The producer and consumer are as follows:

```
process producer;
  var stuff : T;
  loop
    generate stuff;
    send stuff to buffer
    end
  end;
```

[22] Even if message passing is asynchronous, such a buffer may still be required if there are multiple producers or consumers.

```
process consumer;
  var stuff : T;
  loop
    receive stuff from buffer;
    use stuff
    end
  end
```

If **send** statements cannot appear in guards, selective communication is straightforward to implement. A delayed process determines which Boolean expressions in guards are true, and then awaits arrival of a message that allows execution of the **receive** in one of these guards. (If the guard did not contain a **receive**, the process would not be delayed.) If both **send** and **receive** statements can appear in guards,[23] implementation is much more costly because a process needs to negotiate with other processes to determine if they can communicate, and these processes could also be in the middle of such a negotiation. For example, three processes could be executing selective-communications statements in which any pair could communicate; the problem is to decide which pair communicates and which one remains delayed. Development of protocols that solve this problem in an efficient and deadlock-free way remains an active research area [Schwartz, 1978; Silberschatz, 1979; Bernstein, 1980; Van de Snepscheut, 1981; Schneider, 1982; Reif and Spirakis, 1982].

Unfortunately, if **send** statements are not permitted to appear in guards, programming with blocking **send** and blocking **receive** becomes somewhat more complex. In the example above, the *buffer* process above would be changed to first wait for a message from the *consumer* requesting data (a **receive** would appear in the second guard instead of the **send**) and then to send the data. The difference in the protocol used by this new *buffer* process when interacting with the *consumer* and that used when interacting with the *producer* process is misleading; a producer/consumer relationship is inherently symmetric, and the program should mirror this fact.

[23] Also note that allowing only **send** statements in guards is not very useful.

Some process relationships are inherently asymmetric. In client/server interactions, the server often takes different actions in response to different kinds of client requests. For example, a shortest-job-next allocator (see Section 3.4.1) that receives "allocation" requests on a *request_port* and "release" requests on a *release_port* can be programmed using message passing as follows:

```
process shortest_next_allocator;
  var free : Boolean;
      time : integer;
      client_id : process_id;
      declarations of a priority queue and other local variables;
  free := true;
  do true; receive (time, client_id) from request_port →
       if free →
            free := false;
            send allocation to client_id
       [] not free →
            save client_id on priority queue ordered by time
       fi
  [] not free; receive release from release_port →
       if not priority queue empty →
            remove client_id with smallest time from queue;
            send allocation to client_id
       [] priority queue empty →
            free := true
       fi
  od
  end
```

A client makes a request by executing

```
send (time, my_id) to request_port;
receive allocation
    from shortest_next_allocator
```

and indicates that it has finished using the resource by executing

```
send release to release_port
```

4.3 Higher Level Message-Passing Constructs

4.3.1 Remote Procedure Call

The primitives of the previous section are sufficient to program any type of process interaction using message passing. To program client/server interactions, however, both the client and server execute two message-passing statements: the client a **send** followed by a **receive**, and the server a **receive** followed by a **send**. Because this type of interaction is very common, higher level statements that directly support it

have been proposed. These are termed *remote procedure call* statements because of the interface that they present: a client "calls" a procedure that is executed on a potentially remote machine by a server.

When remote procedure calls are used, a client interacts with a server by means of a **call** statement. This statement has a form similar to that used for a procedure call in a sequential language:

call *service*(*value_args*; *result_args*)

The *service* is really the name of a channel. If direct naming is used, *service* designates the server process; if port or mailbox naming is used, *service* might designate the kind of service requested. Remote **call** is executed as follows: the value arguments are sent to the appropriate server, and the calling process delays until both the service has been performed and the results have been returned and assigned to the result arguments. Thus such a **call** could be translated into a **send**, immediately followed by a **receive**. Note that the client cannot forget to wait for the results of a requested service.

There are two basic approaches to specifying the server side of a remote procedure call. In the first, the remote procedure is a declaration, like a procedure in a sequential language:[24]

remote procedure *service*
  (**in** *value_parameters*;
   out *result_parameters*)
 body
 end

However, such a procedure declaration is implemented as a process. This process, the server, awaits receipt of a message containing value arguments from some calling process, assigns them to the value parameters, executes its body, and then returns a *reply message* containing the values of the result parameters. Note that even if there are no value or result parameters, the synchronization resulting from the implicit **send** and **receive** occurs. A remote procedure declaration can be implemented as a single process that repeatedly loops [Andrews, 1982], in which case **calls** to the same remote procedure would execute sequentially. Alternatively, a new process can be created for each execution of **call** [Brinch Hansen, 1978; Cook, 1980; Liskov and Scheifler, 1982]; these could execute concurrently, meaning that the different instances of the server might need to synchronize if they share variables.

In the second approach to specifying the server side, the remote procedure is a statement, which can be placed anywhere any other statement can be placed. Such a statement has the general form

accept *service*(**in** *value_parameters*;
  out *result_parameters*) → *body*

Execution of this statement delays the server until a message resulting from a **call** to the *service* has arrived. Then the body is executed, using the values of the value parameters and any other variables accessible in the scope of the statement. Upon termination, a reply message, containing the values of the result parameters, is sent to the calling process. The server then continues execution.[25]

When **accept** or similar statements are used to specify the server side, remote procedure call is called a *rendezvous* [Department of Defense, 1981] because the client and server "meet" for the duration of the execution of the body of the **accept** statement and then go their separate ways. One advantage of the rendezvous approach is that client **calls** may be serviced at times of the server's choosing; **accept** statements, for example, can be interleaved or nested. A second advantage is that the server can achieve different effects for **calls** to the same service by using more than one **accept** statement, each with a different body. (For example, the first **accept** of a service might perform initialization.) The final, and most important, advantage is that the server can provide more than one kind of service. In particular, **accept** is often combined with selective communications to en-

[24] This is another reason this kind of interaction is termed "remote procedure call."

[25] Different semantics result depending on whether the reply message is sent by a synchronous or by an asynchronous **send**. A synchronous **send** delays the server until the results have been received by the caller. Therefore, when the server continues, it can assert that the reply message has been received and that the result parameters have been assigned to the result arguments. Use of asynchronous **send** does not allow this, but does not delay the server, either.

able a server to wait for and select one of several requests to service [U. S. Department of Defense, 1981; Andrews, 1981]. This is illustrated in the following implementation of the bounded buffer:

```
process buffer;

  var slots : array [0..N-1] of T;
      head, tail : 0..N-1;
      size : 0..N;

  head := 0;  tail := 0;  size := 0;

  do size<N; accept deposit(in value : T) →
       slots[tail] := value;
       size := size + 1;
       tail := (tail + 1) mod N
  [] size>0; accept fetch(out value : T) →
       value := slots[head];
       size := size − 1;
       head := (head + 1) mod N
  od

  end.
```

The *buffer* process implements two operations: *deposit* and *fetch*. The first is invoked by a producer by executing

call *deposit*(*stuff*)

The second is invoked by a consumer by executing

call *fetch*(*stuff*)

Note that *deposit* and *fetch* are handled by the *buffer* process in a symmetric manner, even though **send** statements do not appear in guards, because remote procedure calls always involve two messages, one in each direction. Note also that *buffer* can be used by multiple producers and multiple consumers.

Although remote procedure call is a useful, high-level mechanism for client/server interactions, not all such interactions can be directly programmed by using it. For example, the *shortest_next_allocator* of the previous section still requires two client/server exchanges to service allocation requests because the allocator must look at the parameters of a request in order to decide if the request should be delayed. Thus the client must use one operation to transmit the request arguments and another to wait for an allocation. If there are a small number of different scheduling priorities, this can be overcome by associating a different server operation with each priority level. Ada [U. S. Department of Defense, 1981] supports this nicely by means of arrays of operations. In general, however, a mechanism is required to enable a server to accept a **call** that minimizes some function of the parameters of the called operation. SR [Andrews, 1981] includes such a mechanism (see Section 4.5.4).

4.3.2 Atomic Transactions

An often-cited advantage of multiple-processor systems is that they can be made resilient to failures. Designing programs that exhibit this fault tolerance is not a simple matter. While a discussion of how to design fault-tolerant programs is beyond the scope of this survey, we comment briefly on how fault-tolerance issues have affected the design of higher level message-passing statements.[26]

Remote procedure call provides a clean way to program client/server interactions. Ideally, we would like a remote **call**, like a procedure call in a sequential programming notation, to have *exactly once* semantics: each remote **call** should terminate only after the named remote procedure has been executed exactly once by the server [Nelson, 1981; Spector, 1982]. Unfortunately, a failure may mean that a client is forever delayed awaiting the response to a remote **call**. This might occur if

(i) the message signifying the remote procedure invocation is lost by the network, or
(ii) the reply message is lost, or
(iii) the server crashes during execution of the remote procedure (but before the reply message is sent).

This difficulty can be overcome by attaching a time-out interval to the remote **call**; if no response is received by the client before the time-out interval expires, the client presumes that the server has failed and takes some action.

Deciding what action to take after a detected failure can be difficult. In Case (i) above, the correct action would be to re-

[26] For a general discussion, the interested reader is referred to Kohler 1981.

transmit the message. In Case (ii), however, retransmittal would cause a second execution of the remote procedure body. This is undesirable unless the procedure is *idempotent*, meaning that the repeated execution has the same effect as a single execution. Finally, the correct action in Case (iii) would depend on exactly how much of the remote procedure body was executed, what parts of the computation were lost, what parts must be undone, etc. In some cases, this could be handled by saving state information, called *checkpoints*, and programming special recovery actions. A more general solution would be to view execution of a remote procedure in terms of atomic transactions.

An *atomic transaction* [Lomet, 1977; Reed, 1979; Lampson, 1981] is an all-or-nothing computation—either it installs a complete collection of changes to some variables or it installs no changes, even if interrupted by a failure. Moreover, atomic transactions are assumed to be indivisible in the sense that partial execution of an atomic transaction is not visible to any concurrently executing atomic transaction. The first attribute is called *failure atomicity*, and the second *synchronization atomicity*.

Given atomic transactions, it is possible to construct a remote procedure call mechanism with *at most once* semantics—receipt of a reply message means that the remote procedure was executed exactly once, and failure to receive a reply message means the remote procedure invocation had no (permanent) effect [Liskov and Scheifler, 1982; Spector, 1982]. This is done by making execution of a remote procedure an atomic transaction that is allowed to "commit" only after the reply has ben received by the client. In some circumstances, even more complex mechanisms are useful. For example, when nested remote calls occur, failure while executing a higher level call should cause the effects of lower level (i.e., nested) calls to be undone, even if those calls have already completed [Liskov and Scheifler, 1982].

The main consideration in the design of these mechanisms is that may it not be possible for a process to see system data in an inconsistent state following partial execution of a remote procedure. The use of atomic transactions is one way to do this, but it is quite expensive [Lampson and Sturgis, 1979; Liskov, 1981]. Other techniques to ensure the invisibility of inconsistent states have been proposed [Lynch, 1981; Schlichting and Schneider, 1981], and this remains an active area of research.

4.4 An Axiomatic View of Message Passing

When message passing is used for communication and synchronization, processes usually do not share variables. Nonetheless, interference can still arise. In order to prove that a collection of processes achieves a common goal, it is usually necessary to make assertions in one process about the state of others. Processes learn about each other's state by exchanging messages. In particular, receipt of a message not only causes the transfer of values from sender to receiver but also facilitates the "transfer" of a predicate. This allows the receiver to make assertions about the state of the sender, such as about how far the sender has progressed in its computation. Clearly, subsequent execution by the sender might invalidate such an assertion. Thus it is possible for the sender to interfere with an assertion in the receiver.

It turns out that two distinct kinds of interference must be considered when message passing is used [Schlichting and Schneider, 1982a]. The first is similar to that occurring when shared variables are used: assertions made in one process about the state of another must not be invalidated by concurrent execution. The second form of interference arises only when asynchronous or buffered message passing is used. If a sender "transfers" a predicate with a message, the "transferred" predicate must be true when the message is received: receipt of a message reveals information about the state of the sender at the time that the message was sent, which is not necessarily the sender's current state.

The second type of interference is not possible when synchronous message passing is used, because, after sending a message, the sender does not progress until the

message has been received. This is a good reason to prefer the use of synchronous **send** over asynchronous **send** (and to prefer synchronous **send** for sending the reply message in a remote procedure body). One often hears the argument that asynchronous **send** does not restrict parallelism as much as synchronous **send** and so it is preferable. However, the amount of parallelism that can be exhibited by a program is determined by program structure and not by choice of communications primitives. For example, addition of an intervening buffer process allows the sender to be executed concurrently with the receiving process. Choosing a communications primitive merely establishes whether the programmer will have to do the additional work (of defining more processes) to allow a high degree of parallel activity or will have to do additional work (of using the primitives in a highly disciplined way) to control the amount of parallelism. Nevertheless, a variety of "safe" uses of asynchronous message passing have been identified: the "transfer" of monotonic predicates and the use of "acknowledgment" protocols, for example. These schemes are studied in Schlichting and Schneider [1982b], where they are shown to follow directly from simple techniques to avoid the second kind of interference.

Formal proof techniques for various types of message-passing primitives have been developed. Axioms for buffered, asynchronous message passing were first proposed in connection with Gypsy [Good et al., 1979]. Several people have developed proof systems for synchronous message-passing statements—in particular the input and output commands in CSP [Apt et al., 1980; Cousot and Cousot, 1980; Levin and Gries, 1981; Misra and Chandy, 1981; Soundararajan, 1981; Lamport and Schneider, 1982; Schlichting and Schneider, 1982a]. Also, several people have developed proof rules for asynchronous message passing [Misra et al., 1982; Schlichting and Schneider, 1982b], and proof rules for remote procedures and rendezvous [Barringer and Mearns, 1982; Gerth, 1982; Gerth et al., 1982; Schlichting and Schneider, 1982a].

4.5 Programming Notations Based on Message Passing

A large number of concurrent programming languages have been proposed that use message passing for communication and synchronization. This should not be too surprising; because the two major message-passing design issues—channel naming and synchronization—are orthogonal, the various alternatives for each can be combined in many ways. In the following, we summarize the important characteristics of four languages: CSP, PLITS, Ada, and SR. Each is well documented in the literature and was innovative in some regard. Also, each reflects a different combination of the two design alternatives. Some other languages that have been influential—Gypsy, Distributed Processes, StarMod and Argus—are then briefly discussed.

4.5.1 Communicating Sequential Processes

Communicating Sequential Processes (CSP) [Hoare, 1978] is a programming notation based on synchronous message passing and selective communications. The concepts embodied in CSP have greatly influenced subsequent work in concurrent programming language design and the design of distributed programs.

In CSP, processes are denoted by a variant of the **cobegin** statement. Processes may share read-only variables, but use input/output commands for synchronization and communication. Direct (and static) channel naming is used and message passing is synchronous.

An *output command* in CSP has the form

destination!*expression*

where *destination* is a process name and *expression* is a simple or structured value. An *input command* has the form

source?*target*

where *source* is a process name and *target* is a simple or structured variable local to the process containing the input command. The commands

Pr!*expression*

in process *Ps* and

Ps?*target*

in process *Pr match* if *target* and *expression* have the same type. Two processes communicate if they execute a matching pair of input/output commands. The result of communication is that the expression's value is assigned to the target variable; both processes then proceed independently and concurrently.

A restricted form of selective communications statement is supported by CSP. Input commands can appear in guards of alternative and iterative statements, but output commands may not. This allows an efficient implementation, but makes certain kinds of process interaction awkward to express, as was discussed in Section 4.2.

By combining communication commands with alternative and iterative statements, CSP provides a powerful mechanism for programming process interaction. Its strength is that it is based on a simple idea—input/output commands—that is carefully integrated with a few other mechanisms. CSP is not a complete concurrent programming language, nor was it intended to be. For example, static direct naming is often awkward to use. Fortunately, this deficiency is easily overcome by using ports; how to do so was discussed briefly by Hoare [Hoare, 1978] and is described in detail by Kieburtz and Silberschatz [1979]. Recently, two languages based on CSP have also been described [Jazayeri et al., 1980; Roper and Barter, 1981].

4.5.2 PLITS

PLITS, an acronym for "Programming Language In The Sky," was developed at the University of Rochester [Feldman, 1979]. The design of PLITS is based on the premise that it is inherently difficult to combine a high degree of parallelism with data sharing and therefore message passing is the appropriate means for process interaction in a distributed system. Part of an ongoing research project in programming language design and distributed computation, PLITS is being used to program applications that are executed on Rochester's Intelligent Gateway (RIG) computer network [Ball et al., 1976].

A PLITS program consists of a number of modules; *active modules* are processes. Message passing is the sole means for intermodule interaction. So as not to restrict parallelism, message passing is asynchronous. A module sends a message containing the values of some expressions to a module *modname* by executing

send *expressions* **to** *modname* [**about** *key*]

The "**about** *key*" phrase is optional. If included, it attaches an identifying *transaction key* to the message. This key can then be used to identify the message uniquely, or the same key can be attached to several different messages to allow messages to be grouped.

A module receives messages by executing

receive *variables* [**from** *modname*]
  [**about** *key*]

If the last two phrases are omitted, execution of **receive** delays the executing module until the arrival of any message. If the phrase "**from** *modname*" is included, execution is delayed until a message from the named module arrives. Finally, if the phrase "**about** *key*" is included, the module is delayed until a message with the indicated transaction key has arrived.

By combining the options in **send** and **receive** in different ways, a programmer can exert a variety of controls over communication. When both the sending and receiving modules name each other, communication is direct. The effect of port naming is realized by having a receiving module not name the source module. Finally, the use of transaction keys allows the receiver to select a particular kind of message; this provides a facility almost as powerful as attaching "**when** *B*" to a **receive** statement.

In PLITS, execution of **receive** can cause blocking. PLITS also provides primitives to test whether messages with certain field values or transaction keys are available for receipt; this enables a process to avoid blocking when there is no message available.

PLITS programs interface to the operating systems of the processors that make up RIG. Each host system provides device access, a file system, and job control. A

communications kernel on each machine provides the required support for interprocessor communication.

4.5.3 Ada[27]

Ada [U. S. Department of Defense, 1981] is a language intended for programming embedded real-time, process-control systems. Because of this, Ada includes facilities for multiprocessing and device control. With respect to concurrent programming, Ada's main innovation is the rendezvous form of remote procedure call.

Processes in Ada are called *tasks*. A task is activated when the block containing its declaration is entered. Tasks may be nested and may interact by using shared variables declared in enclosing blocks. (No special mechanisms for synchronizing access to shared variables are provided.)

The primary mechanism for process interaction is the remote procedure call. Remote procedures in Ada are called *entries*; they are ports into a server process specified by means of an **accept** statement, which is similar in syntax and semantics to the **accept** statement described in Section 4.3.1. Entries are invoked by execution of a remote **call**. Selective communications is supported using the **select** statement, which is like an alternative statement.

Both **call** and **accept** statements are blocking. Since Ada programs might have to meet real-time response constraints, the language includes mechanisms to prevent or control the length of time that a process is delayed when it becomes blocked. Blocking on **call** can be avoided by using the *conditional entry call*, which performs a **call** only if a rendezvous is possible immediately. Blocking on **accept** can be avoided by using a mechanism that enables a server to determine the number of waiting **calls**. Blocking on **select** can be avoided by means of the **else** guard, which is true if none of the other guards are. Finally, a task can suspend execution for a time interval by means of the **delay** statement. This statement can be used within a guard of **select** to ensure that a process is eventually awakened.

In order to allow the programmer to control I/O devices, Ada allows entries to be bound to interrupt vector locations. Interrupts become calls to those entries and can therefore be serviced by a task that receives the interrupt by means of an **accept** statement.

Since its inception, Ada has generated controversy [Hoare, 1981], much of which is not related to concurrency. However, few applications using the concurrent programming features have been programmed, and at the time of this writing no compiler for full Ada has been validated. Implementation of some of the concurrent programming aspects of Ada is likely to be hard. A paper by Welsh and Lister [1981] compares the concurrency aspects of Ada to CSP and Distributed Processes [Brinch Hansen, 1978]; Wegner and Smolka [1983] compare Ada, CSP, and monitors.

4.5.4 SR

SR (Synchronizing Resources) [Andrews, 1981, 1982], like Ada, uses the rendezvous form of remote procedure call and port naming. However, there are notable differences between the languages, as described below. A compiler for SR has been implemented on PDP-11 processors and the language is being used in the construction of a UNIX-like network operating system.

An SR program consists of one or more *resources*.[28] The resource construct supports both control of process interaction and data abstraction. (In contrast, Ada has two distinct constructs for this—the task and the package.) Resources contain one or more processes. Processes interact by using *operations*, which are similar to Ada entries. Also, processes in the same resource may interact by means of shared variables.

Unlike Ada, operations may be invoked by either **send**, which is nonblocking, or **call**, which is blocking. (The server that implements an operation can require a particular form of invocation, if necessary.) Thus both asynchronous message passing and remote **call** are supported. Operations may be named either statically in the program text or dynamically by means of ca-

[27] Ada is a trademark of the U. S. Department of Defense.

[28] SR's resources are not to be confused with resources in conditional critical regions.

pability variables, which are variables having fields whose values are the names of operations. A process can therefore have a changing set of communication channels.

In SR, operations are specified by the **in** *statement*, which also supports selective communications. Each guard in an **in** statement has the form

op_name(*parameters*) [**and** B] [**by** A]

where B is an optional Boolean expression and A is an optional arithmetic expression. The phrase "**and** B" allows selection of the operation to be dependent on the value of B, which may contain references to parameters. The phrase "**by** A" controls which invocation of *op_name* is selected if more than one invocation is pending that satisfies B. This can be used to express scheduling constraints succinctly. For example, it permits a compact solution to the shortest-job-next allocation problem discussed earlier. Although somewhat expensive to implement because it requires reevaluation of A whenever a selection is made, this facility turns out to be less costly to use than explicitly programmed scheduling queues, if the expected number of pending invocations is small (which is usually the case).

Operations may also be declared to be *procedures*. In SR, a procedure is shorthand for a process that repeatedly executes an **in** statement. Thus such operations are executed sequentially.

To support device control, SR provides a variant of the resource called a *real resource*. A real resource is similar to a Modula device module: it can contain device-driver processes and it allows variables to be bound to device-register addresses. Operations in real resources can be bound to interrupt vector locations. A hardware interrupt is treated as a **send** to such an operation; interrupts are processed by means of **in** statements.

4.5.5 Some Other Language Notations Based on Message Passing

Gypsy [Good et al., 1979], one of the first high-level languages based on message passing, uses mailbox naming and buffered message passing. A major focus of Gypsy was the development of a programming language well suited for constructing verifiable systems. It has been used to implement special-purpose systems for single- and multiprocessor architectures.

Distributed Processes (DP) [Brinch Hansen, 1978] was the first language to be based on remote procedure calls. It can be viewed as a language that implements monitors by means of active processes rather than collections of passive procedures. In DP, remote procedures are specified as externally callable procedures declared along with a host process and shared variables. When a remote procedure is called, a server process is created to execute the body of the procedure. The server processes created for different calls and the host process execute with mutual exclusion. The servers and host synchronize by means of a variant of conditional critical regions. An extension of DP that employs the rendezvous form of remote procedure call and thus has a more efficient implementation is described by Mao and Yeh [1980].

StarMod [Cook, 1980] synthesizes aspects of Modula and Distributed Processes: it borrows modularization ideas from Modula and communication ideas from Distributed Processes. A module contains one or more processes and, optionally, variables shared by those processes. Synchronization within a module is provided by semaphores. Processes in different modules interact by means of remote procedure call; StarMod provides both remote procedures and rendezvous for implementing the server side. In StarMod, as in SR, both **send** and **call** can be used to initiate communication, the choice being dictated by whether the invoked operation returns values.

Argus [Liskov and Scheifler, 1982] also borrows ideas from Distributed Processes—remote procedures implemented by dynamically created processes, which synchronize using critical regions—but goes much further. It has extensive support for programming atomic transactions. The language also includes exception handling and recovery mechanisms, which are invoked if failures occur during execution of atomic transactions. Argus is higher level than the other languages surveyed here in the sense that it attaches more semantics to remote **call**. A prototype implementation of Argus is nearing completion.

5. MODELS OF CONCURRENT PROGRAMMING LANGUAGES

Most of this survey has been devoted to mechanisms for process interaction and programming languages that use them. Despite the resulting large variety of languages, each can be viewed as belonging to one of three classes: procedure oriented, message oriented, or operation oriented. Languages in the same class provide the same basic kinds of mechanisms for process interaction and have similar attributes.

In *procedure-oriented* languages, process interaction is based on shared variables. (Because monitor-based languages are the most widely known languages in this class, this is often called the *monitor model.*) These languages contain both active objects (processes) and shared, passive objects (modules, monitors, etc.). Passive objects are represented by shared variables, usually with some procedures that implement the operations on the objects. Processes access the objects they require directly and thus interact by accessing shared objects. Because passive objects are shared, they are subject to concurrent access. Therefore, procedure-oriented languages provide means for ensuring mutual exclusion. Concurrent PASCAL, Modula, Mesa, and Edison are examples of such languages.

Message- and operation-oriented languages are both based on message passing, but reflect different views of process interaction. *Message-oriented* languages provide **send** and **receive** as the primary means for process interaction. In contrast to procedure-oriented languages, there are no shared, passive objects, and so processes cannot directly access all objects. Instead, each object is managed by a single process, its *caretaker*, which performs all operations on it. When an operation is to be performed on an object, a message is sent to its caretaker, which performs the operation and then (possibly) responds with a completion message. Thus, objects are never subject to concurrent access. CSP, Gypsy, and PLITS are examples of message-oriented languages.

Operation-oriented languages provide remote procedure call as the primary means for process interaction. These languages combine aspects of the other two classes. As in a message-oriented language, each object has a caretaker process associated with it; as in a procedure-oriented language, operations are performed on an object by calling a procedure. The difference is that the caller of an operation and the caretaker that implements it synchronize while the operation is executed. Both then proceed asynchronously. Distributed Processes, StarMod, Ada, and SR are examples of operation-oriented languages.

Languages in each of these classes are roughly equivalent in expressive power. Each can be used to implement various types of cooperation between concurrently executing processes, including client/server interactions and pipelines. Operation-oriented languages are well suited for programming client/server systems, and message-oriented languages are well suited for programming pipelined computations.

Languages in each class can be used to write concurrent programs for uniprocessors, multiprocessors, and distributed systems. Not all three classes are equally suited for all three architectures, however. Procedure-oriented languages are the most efficient to implement on contemporary single processors. Since it is expensive to simulate shared memory if none is present, implementing procedure-oriented languages on a distributed system can be costly. Nevertheless, procedure-oriented languages can be used to program a distributed system—an individual program is written for each processor and the communications network is viewed as a shared object. Message-oriented languages can be implemented with or without shared memory. In the latter case, the existence of a communications network is made completely transparent, which frees the programmer from concerns about how the network is accessed and where processes are located. This is an advantage of message-oriented languages over procedure-oriented languages when programming a distributed system. Operation-oriented languages enjoy the advantages of both procedure-oriented and message-oriented languages. When shared memory is available, an operation-oriented language can, in many cases, be implemented like a procedure-oriented language [Habermann and Nassi,

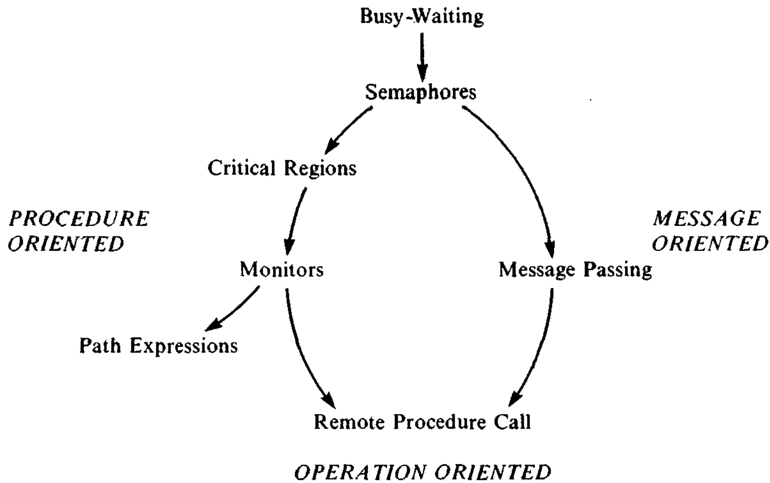

Figure 9. Synchronization techniques and language classes.

1980]; otherwise it can be implemented using message passing. Recent research has shown that both message- and operation-oriented languages can be implemented quite efficiently on distributed systems if special software/firmware is used in the implementation of the language's mechanisms [Nelson, 1981; Spector, 1982].

In a recent paper, Lauer and Needham argued that procedure-oriented and message-oriented languages are equals in terms of expressive power, logical equivalence, and performance [Lauer and Needham, 1979]. (They did not consider operation-oriented languages, which have only recently come into existence.) Their thesis was examined in depth by Reid [1980], who reached many conclusions that we share. At an abstract level, the three types of languages are interchangeable. One *can* transform any program written using the mechanisms found in languages of one class into a program using the mechanisms of another class without affecting performance. However, the classes emphasize different styles of programming—the same program written in languages of different classes is often best structured in entirely different ways. Also, each class provides a type of flexibility not present in the others. Program fragments that are easy to describe using the mechanisms of one can be awkward to describe using the mechanisms of another. One might argue (as do Lauer and Needham) that such use of these mechanisms is a bad idea. We, however, favor programming in the style appropriate to the language.

6. CONCLUSION

This paper has discussed two aspects of concurrent programming: the key concepts—specification of processes and control of their interaction—and important language notations. Early work on operating systems led to the discovery of two types of synchronization: mutual exclusion and condition synchronization. This stimulated development of synchronization primitives, a number of which are described in this paper. The historical and conceptual relationships among these primitives are illustrated in Figure 9.

The difficulty of designing concurrent programs that use busy-waiting and their inefficiency led to the definition of semaphores. Semaphores were then extended in two ways: (1) constructs were defined that enforced their structured use, resulting in critical regions, monitors, and path expressions; (2) "data" were added to the synchronization associated with semaphores, resulting in message-passing primitives. Finally, the procedural interface of monitors was combined with message passing, resulting in remote procedure call.

Since the first concurrent programming languages were defined only a decade ago, practical experience has increased our un-

derstanding of how to engineer such programs, and the development of formal techniques has greatly increased our understanding of the basic concepts. Although there are a variety of different programming languages, there are only three essentially different kinds: procedure oriented, message oriented, and operation oriented. This, too, is illustrated in Figure 9.

At present, many of the basic problems that arise when constructing concurrent programs have been identified, solutions to these problems are by and large understood, and substantial progress has been made toward the design of notations to express those solutions. Much remains to be done, however. The utility of various languages—really, combinations of constructs—remains to be investigated. This requires using the languages to develop systems and then analyzing how they helped or hindered the development. In addition, the interaction of fault tolerance and concurrent programming is not well understood. Little is known about the design of distributed (decentralized) concurrent programs. Last, devising formal techniques to aid the programmer in constructing correct programs remains an important open problem.

ACKNOWLEDGMENTS

Numerous people have been kind enough to provide very helpful comments on earlier drafts of this survey: David Gries, Phil Kaslo, Lynn Kivell, Gary Levin, Ron Olsson, Rick Schlichting, and David Wright. Three referees, and also Eike Best and Michael Scott, provided valuable comments on the penultimate draft. Tony Wasserman has also provided helpful advice; it has been a joy to have him as the editor for this paper. Rachel Rutherford critiqued the ultimate draft and made numerous useful, joyfully picturesque comments.

This work was supported in part by NSF Grants MCS 80-01668 and MCS 82-02869 at Arizona and MCS 81-03605 at Cornell.

REFERENCES

AKKOYUNLU, E. A., BERNSTEIN, A. J., SCHNEIDER, F. B., AND SILBERSCHATZ, A. "Conditions for the equivalence of synchronous and asynchronous systems." *IEEE Trans. Softw. Eng.* **SE-4,** 6 (Nov. 1978), 507-516.

ANDLER, S. "Predicate path expressions." In *Proc. 6th ACM Symp. Principles of Programming Languages* (San Antonio, Tex., Jan. 1979). ACM, New York, 1979, pp. 226-236.

ANDREWS, G. R. "The design of a message switching system: An application and evaluation of Modula." *IEEE Trans. Softw. Eng.* **SE-5,** 2 (March 1979), 138-147.

ANDREWS, G. R. "Synchronizing resources." *ACM Trans. Prog. Lang. Syst.* **3,** 4 (Oct. 1981), 405-430.

ANDREWS, G. R. "The distributed programming language SR—Mechanisms, design, and implementation." *Softw. Pract. Exper.* **12,** 8 (Aug. 1982), 719-754.

ANDREWS, G. R., AND MCGRAW, J. R. "Language features for process interaction." In *Proc. ACM Conf. Language Design for Reliable Software, SIGPLAN Not.* **12,** 3 (March 1977), 114-127.

APT, K. R., FRANCEZ, N., AND DE ROEVER, W. P. "A proof system for communicating sequential processes." *ACM Trans. Prog. Lang. Syst.* **2,** 3 (July 1980), 359-385.

ASCHCROFT, E. A. "Proving assertions about parallel programs." *J. Comput. Syst.* **10** (Jan. 1975), 110-135.

BALL, E., FELDMAN, J., LOW, J., RASHID, R., AND ROVNER, P. "RIG, Rochester's intelligent gateway: System overview." *IEEE Trans. Softw. Eng.* **SE-2,** 4 (Dec. 1976), 321-328.

BALZER, R. M. "PORTS—A method for dynamic interprogram communication and job control." In *Proc. AFIPS Spring Jt. Computer Conf.* (Atlantic City, N. J., May 18-20, 1971), vol. 38. AFIPS Press, Arlington, Va., 1971, pp. 485-489.

BARRINGER, H., AND MEARNS, I. "Axioms and proof rules for Ada tasks." *IEE Proc.* **129,** Pt. E, 2 (March 1982), 38-48.

BASKETT, F., HOWARD, J. H., AND MONTAGUE, J. T. "Task communication in DEMOS." In *Proc. 6th Symp. Operating Systems Principles* (West Lafayette, Indiana, Nov. 16-18, 1977). ACM, New York, 1977, pp. 23-31.

BEN-ARI, M. *Principles of Concurrent Programming.* Prentice-Hall, Englewood Cliffs, N. J., 1982.

BERNSTEIN, A. J. "Output guards and nondeterminism in communicating sequential processes." *ACM Trans. Prog. Lang. Syst.* **2,** 2 (Apr. 1980), 234-238.

BERNSTEIN, A. J., AND ENSOR, J. R. "A modification of Modula." *Softw. Pract. Exper.* **11** (1981), 237-255.

BERNSTEIN, A. J., AND SCHNEIDER, F. B. "On language restrictions to ensure deterministic behavior in concurrent systems." In J. Moneta (Ed.), *Proc. 3rd Jerusalem Conf. Information Technology JCIT3.* North-Holland Publ., Amsterdam, 1978, pp. 537-541.

BERNSTEIN, P. A., AND GOODMAN, N. "Concurrency control in distributed database systems." *ACM Comput. Surv.* **13,** 2 (June 1981), 185-221.

BEST, E. "Relational semantics of concurrent programs (with some applications)." In *Proc. IFIP WG2.2 Conf.* North-Holland Publ., Amsterdam, 1982.

BLOOM, T. "Evaluating synchronization mechanisms." In *Proc. 7th Symp. Operating Systems Principles* (Pacific Grove, Calif., Dec. 10-12, 1979). ACM, New York, 1979, pp. 24-32.

BRINCH HANSEN, P. "Structured multiprogramming." *Commun. ACM* **15**, 7 (July 1972), 574-578.

BRINCH HANSEN, P. *Operating System Principles*. Prentice-Hall, Englewood Cliffs, N. J., 1973. (a)

BRINCH HANSEN, P. "Concurrent programming concepts." *ACM Comput. Surv.* **5**, 4 (Dec. 1973), 223-245. (b)

BRINCH HANSEN, P. "The programming language Concurrent Pascal." *IEEE Trans Softw. Eng.* **SE-1**, 2 (June 1975), 199-206.

BRINCH HANSEN, P. "The Solo operating system: Job interface." *Softw. Pract. Exper.* **6** (1976), 151-164. (a)

BRINCH HANSEN, P. "The Solo operating system: Processes, monitors, and classes." *Softw. Pract. Exper.* **6** (1976), 165-200. (b)

BRINCH HANSEN, P. *The Architecture of Concurrent Programs*. Prentice-Hall, Englewood Cliffs, N. J., 1977.

BRINCH HANSEN, P. "Distributed processes: A concurrent programming concept." *Commun. ACM* **21**, 11 (Nov. 1978), 934-941.

BRINCH HANSEN, P. "Edison: A multiprocessor language." *Softw. Pract. Exper.* **11**, 4 (Apr. 1981), 325-361.

CAMPBELL, R. H. "Path expressions: A technique for specifying process synchronization." Ph.D. dissertation, Computing Laboratory, University of Newcastle upon Tyne, Aug. 1976.

CAMPBELL, R. H., AND HABERMANN, A. N. "The specification of process synchronization by path expressions." *Lecture Notes in Computer Science*, vol. 16. Springer-Verlag, New York, 1974, pp. 89-102.

CAMPBELL, R. H., AND KOLSTAD, R. B. "Path expressions in Pascal." In *Proc. 4th Int. Conf. on Software Eng.* (Munich, Sept. 17-19, 1979). IEEE, New York, 1979, pp. 212-219.

CONWAY, M. E. "Design of a separable transition-diagram compiler." *Commun. ACM* **6**, 7 (July 1963), 396-408. (a)

CONWAY, M. E. "A multiprocessor system design." In *Proc. AFIPS Fall Jt. Computer Conf.* (Las Vegas, Nev., Nov., 1963), vol. 24. Spartan Books, Baltimore, Maryland, pp. 139-146. (b)

COOK, R. P. "*MOD—A language for distributed programming." *IEEE Trans. Softw. Eng.* **SE-6**, 6 (Nov. 1980), 563-571.

COURTOIS, P. J., HEYMANS, F., AND PARNAS, D. L. "Concurrent control with 'readers' and 'writers'." *Commun. ACM* **14**, 10 (Oct. 1971), 667-668.

COUSOT, P., AND COUSOT, R. "Semantic analysis of communicating sequential processes." In *Proc. 7th Int. Colloquium Automata, Languages and Programming (ICALP80), Lecture Notes in Computer Science*, vol. 85. Springer-Verlag, New York, 1980, pp. 119-133.

DENNIS, J. B., AND VAN HORN, E. C. "Programming semantics for multiprogrammed computations." *Commun. ACM* **9**, 3 (March 1966), 143-155.

DIJKSTRA, E. W. "The structure of the 'THE' multiprogramming system." *Commun. ACM* **11**, 5 (May 1968), 341-346. (a)

DIJKSTRA, E. W. "Cooperating sequential processes." In F. Genuys (Ed.), *Programming Languages*. Academic Press, New York, 1968. (b)

DIJKSTRA, E. W. "Guarded commands, nondeterminacy, and formal derivation of programs." *Commun. ACM* **18**, 8 (Aug. 1975), 453-457.

DIJKSTRA, E. W. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N. J., 1976.

DIJKSTRA, E. W. "An assertional proof of a program by G. L. Peterson." EWD 779 (Feb. 1979), Nuenen, The Netherlands. (a)

DIJKSTRA, E. W. Personal communication, Oct. 1981. (b)

FELDMAN, J. A.. "High level programming for distributed computing." *Commun. ACM* **22**, 6 (June 1979), 353-368.

FLON, L., AND HABERMANN, A. N. "Towards the construction of verifiable software systems." In *Proc. ACM Conf. Data, SIGPLAN Not.* **8**, 2 (March 1976), 141-148.

FLOYD, R. W. "Assigning meanings to programs." In *Proc. Am. Math. Soc. Symp. Applied Mathematics*, vol. 19, pp. 19-31, 1967.

GELERNTER, D., AND BERNSTEIN, A. J. "Distributed communication via global buffer." In *Proc. Symp. Principles of Distributed Computing* (Ottawa, Canada, Aug. 18-20, 1982). ACM, New York, 1982, pp. 10-18.

GERTH, R. "A sound and complete Hoare axiomatization of the Ada-rendezvous." In *Proc. 9th Int. Colloquium Automata, Languages and Programming (ICALP82), Lecture Notes in Computer Science*, vol. 140. Springer-Verlag, New York, 1982, pp. 252-264.

GERTH, R., DE ROEVER, W. P., AND RONCKEN, M. "Procedures and concurrency: A study in proof." In *5th Int. Symp. Programming, Lecture Notes in Computer Science*, vol. 137. Springer-Verlag, New York, 1982, pp. 132-163.

GOOD, D. I., COHEN, R. M., AND KEETON-WILLIAMS, J. "Principles of proving concurrent programs in Gypsy." In *Proc. 6th ACM Symp. Principles of Programming Languages* (San Antonio, Texas, Jan. 29-31, 1979). ACM, New York, 1979, pp. 42-52.

HABERMANN, A. N. "Path expressions." Dep. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pennsylvania, June, 1975.

HABERMANN, A. N., AND NASSI, I. R. "Efficient implementation of Ada tasks." Tech. Rep. CMU-CS-80-103, Carnegie-Mellon Univ., Jan. 1980.

HADDON, B. K. "Nested monitor calls." *Oper. Syst. Rev.* **11**, 4 (Oct. 1977), 18-23.

HANSON, D. R., AND GRISWOLD, R. E. "The SL5 procedure mechanism." *Commun. ACM* **21**, 5 (May 1978), 392-400.

HOARE, C. A. R. "An axiomatic basis for computer programming." *Commun. ACM.* **12,** 10 (Oct. 1969), 576-580, 583.

HOARE, C. A. R. "Towards a theory of parallel programming." In C. A. R. Hoare and R. H. Perrott (Eds.), *Operating Systems Techniques.* Academic Press, New York, 1972, pp. 61-71.

HOARE, C. A. R. "Monitors: An operating system structuring concept." *Commun. ACM* **17,** 10 (Oct. 1974), 549-557.

HOARE, C. A. R. "Communicating sequential processes." *Commun. ACM* **21,** 8 (Aug. 1978), 666-677.

HOARE, C. A. R. "The emperor's old clothes." *Commun. ACM* **24,** 2 (Feb. 1981), 75-83.

HOLDEN, J., AND WAND, I. C. "An assessment of Modula." *Softw. Pract. Exper.* **10** (1980), 593-621.

HOLT, R. C., GRAHAM, G. S., LAZOWSKA, E. D., AND SCOTT, M. A. *Structured Concurrent Programming with Operating Systems Applications.* Addison-Wesley, Reading, Mass., 1978.

HOWARD, J. H. "Proving monitors." *Commun. ACM* **19,** 5 (May 1976), 273-279. (a)

HOWARD, J. H. "Signaling in monitors." In *Proc. 2nd Int. Conf. Software Engineering* (San Francisco, Oct. 13-15, 1976). IEEE, New York, 1976, pp. 47-52. (b)

JAZAYERI, M., et al. "CSP/80: A language for communicating processes." In *Proc. Fall IEEE COMPCON80* (Sept. 1980). IEEE, New York, 1980, pp. 736-740.

JONES, A. K., AND SCHWARZ, P. "Experience using multiprocessor systems—A status report." *ACM Comput. Surv.* **12,** 2 (June 1980), 121-165.

KAUBISCH, W. H., PERROTT, R. H., AND HOARE, C. A. R. "Quasiparallel programming." *Softw. Pract. Exper.* **6** (1976), 341-356.

KEEDY, J. L. "On structuring operating systems with monitors." *Aust. Comput. J.* **10,** 1 (Feb. 1978), 23-27. Reprinted in *Oper. Syst. Rev.* **13,** 1 (Jan. 1979), 5-9.

KELLER, R. M. "Formal verification of parallel programs." *Commun. ACM* **19,** 7 (July 1976), 371-384.

KESSELS, J. L. W. "An alternative to event queues for synchronization in monitors." *Commun. ACM* **20,** 7 (July 1977), 500-503.

KIEBURTZ, R. B., AND SILBERSCHATZ, A. "Comments on 'communicating sequential processes.'" *ACM Trans. Program. Lang. Syst.* **1,** 2 (Oct. 1979), 218-225.

KOHLER, W. H. "A survey of techniques for synchronization and recovery in decentralized computer systems." *ACM Comput. Surv.* **13,** 2 (June 1981), 149-183.

LAMPORT, L. "Proving the correctness of multiprocess programs." *IEEE Trans. Softw. Eng.* **SE-3,** 2 (March 1977), 125-143.

LAMPORT, L. "The 'Hoare logic' of concurrent programs." *Acta Inform.* **14,** 21-37. (a)

LAMPORT, L. "The mutual exclusion problem." Op. 56, SRI International, Menlo Park, Calif., Oct. 1980. (b)

LAMPORT, L., AND SCHNEIDER, F. B. "The 'Hoare logic' of CSP, and all that." Tech. Rep. TR 82-490, Dep. Computer Sci., Cornell Univ., May, 1982.

LAMPSON, B. W. "Atomic transactions." In *Distributed Systems—Architecture and Implementation, Lecture Notes in Computer Science,* vol. 105. Springer-Verlag, New York, 1981.

LAMPSON, B. W., AND REDELL, D. D. "Experience with processes and monitors in Mesa." *Commun. ACM* **23,** 2 (Feb. 1980), 105-117.

LAMPSON, B. W., AND STURGIS, H. E. "Crash recovery in a distributed data storage system." Xerox Palo Alto Research Center, Apr. 1979.

LAUER, H. C., AND NEEDHAM, R. M. "On the duality of operating system structures." In *Proc. 2nd Int. Symp. Operating Systems* (IRIA, Paris, Oct. 1978); reprinted in *Oper. Syst. Rev.* **13,** 2 (Apr. 1979), 3-19.

LAUER, P. E., AND CAMPBELL, R. H. "Formal semantics of a class of high level primitives for coordinating concurrent processes." *Acta Inform.* **5** (1975), 297-332.

LAUER, P. E., AND SHIELDS, M. W. "Abstract specification of resource accessing disciplines: Adequacy, starvation, priority and interrupts." *SIGPLAN Not.* **13,** 12 (Dec. 1978), 41-59.

LEHMANN, D., PNUELI, A., AND STAVI, J. "Impartiality, justice and fairness: The ethics of concurrent termination." *Automata, Languages and Programming, Lecture Notes in Computer Science,* vol. 115. Springer-Verlag, New York, 1981, pp. 264-277.

LEVIN, G. M., AND GRIES, D. "A proof technique for communicating sequential processes." *Acta Inform.* **15** (1981), 281-302.

LISKOV, B. L. "On linguistic support for distributed programs." In *Proc. IEEE Symp. Reliability in Distributed Software and Database Systems* (Pittsburgh, July 21-22, 1981). IEEE, New York, 1981, pp. 53-60.

LISKOV, B. L., AND SCHEIFLER, R. "Guardians and actions: Linguistic support for robust, distributed programs." In *Proc. 9th ACM Symp. Principles of Programming Languages* (Albuquerque, New Mexico, Jan. 25-27, 1982). ACM, New York, 1982, pp. 7-19.

LISTER, A. "The problem of nested monitor calls." *Oper. Syst. Rev.* **11,** 3 (July 1977), 5-7.

LOEHR, K.-P. "Beyond Concurrent Pascal." In *Proc. 6th ACM Symp. Operating Systems Principles* (West Lafayette, Ind., Nov. 16-18, 1977). ACM, New York, 1977, pp. 173-180.

LOMET, D. B. "Process structuring, synchronization, and recovery using atomic transactions." In *Proc. ACM Conf. Language Design for Reliable Software, SIGPLAN Not.* **12,** 3 (March 1977), 128-137.

LYNCH, N. A. "Multilevel atomicity—A new correctness criterion for distributed databases." Tech.

Rep. GIT-ICS-81/05, School of Information and Computer Sciences, Georgia Tech., May 1981.

MAO, T. W., AND YEH, R. T. "Communication port: A language concept for concurrent programming." *IEEE Trans. Softw. Eng.* **SE-6**, 2 (March 1980), 194–204.

MISRA, J., AND CHANDY, K. "Proofs of networks of processes." *IEEE Trans. Softw. Eng.* **SE-7**, 4 (July 1981), 417–426.

MISRA, J., CHANDY, K. M., AND SMITH, T. "Proving safety and liveness of communicating processes with examples." In *Proc. Symp. Principles of Distributed Computing* (Ottawa, Canada, Aug. 18–20, 1982). ACM, New York, 1982, pp. 201–208.

MITCHELL, J. G., MAYBURY, W., AND SWEET, R. "Mesa language manual, version 5.0." Rep. CSL-79-3, Xerox Palo Alto Research Center, Apr. 1979.

NELSON, B. J. "Remote procedure call." Ph.D. thesis. Rep. CMU-CS-81-119, Dep. of Computer Science, Carnegie-Mellon Univ., May 1981.

NYGAARD, K., AND DAHL, O. J. "The development of the SIMULA languages." *Preprints ACM SIGPLAN History of Programming Languages Conference, SIGPLAN Not.* **13**, 8 (Aug. 1978), 245–272.

OWICKI, S. S., AND GRIES, D. "An axiomatic proof technique for parallel programs." *Acta Inform.* **6** (1976), 319–340. (a)

OWICKI, S. S., AND GRIES, D. "Verifying properties of parallel programs: an axiomatic approach." *Commun. ACM* **19**, 5 (May 1976), 279–285. (b)

PARNAS, D. L. "On the criteria to be used in decomposing systems into modules." *Commun. ACM* **15**, 12 (Dec. 1972), 1053–1058.

PARNAS, D. L. "The non-problem of nested monitor calls." *Oper. Syst. Rev.* **12**, 1 (Jan. 1978), 12–14.

PETERSON, G. L. "Myths about the mutual exclusion problem." *Inform. Process. Lett.* **12**, 3 (June 1981), 115–116.

REED, D. P. "Implementing atomic actions on decentralized data." *ACM Trans. Comput. Syst.* **1**, 1 (Feb. 1983), 3–23.

REID, L. G. "Control and communication in programmed systems." Ph.D. thesis, Rep. CMU-CS-80-142, Dep. of Computer Science, Carnegie-Mellon Univ., Sept. 1980.

REIF, J. H., AND SPIRAKIS, P. G. "Unbounded speed variability in distributed communications systems." In *Proc. 9th ACM Conf. Principles of Programming Languages* (Albuquerque, N. M., Jan. 25–27, 1982). ACM, New York, 1982, pp. 46–56.

RITCHIE, D. M., AND THOMPSON, K. "The UNIX timesharing system. *Commun. ACM* **17**, 7 (July 1974), 365–375.

ROPER, T. J., AND BARTER, C. J. "A communicating sequential process language and implementation." *Softw. Pract. Exper.* **11** (1981), 1215–1234.

SCHLICHTING, R. D., AND SCHNEIDER, F. B. "An approach to designing fault-tolerant computing systems." Tech. Rep. TR 81-479, Dep. of Computer Sci., Cornell Univ., Nov. 1981.

SCHLICHTING, R. D., AND SCHNEIDER, F. B. "Using message passing for distributed programming: Proof rules and disciplines." Tech. Rep. TR 82-491, Dep. of Computer Science, Cornell Univ., May 1982. (a)

SCHLICHTING, R. D., AND SCHNEIDER, F. B. "Understanding and using asynchronous message passing primitives." In *Proc. Symp. Principles of Distributed Computing* (Ottawa, Canada, Aug. 18–20, 1982). ACM, New York, 1982, pp. 141–147. (b)

SCHNEIDER, F. B. "Synchronization in distributed programs." *ACM Trans. Program. Lang. Syst.* **4**, 2 (Apr. 1982), 125–148.

SCHNEIDER, F. B., AND BERNSTEIN, A. J. "Scheduling in Concurrent Pascal." *Oper. Syst. Rev.* **12**, 2 (Apr. 1978), 15–20.

SCHWARTZ, J. S. "Distributed synchronization of communicating sequential processes." Tech. Rep., Dep. of Artificial Intelligence, Univ. of Edinburgh, July 1978.

SHAW, A. C. *The Logical Design of Operating Systems.* Prentice-Hall, Englewood Cliffs, N. J., 1974.

SHAW, A. C. "Software specification languages based on regular expressions." In W. E. Riddle and R. E. Fairley (Eds.), *Software Development Tools.* Springer-Verlag, New York, 1980, pp. 148–175.

SHIELDS, M. W. "Adequate path expressions." In *Proc. Int. Symp. Semantics of Concurrent Computation, Lecture Notes in Computer Science,* vol. 70. Springer-Verlag, New York, pp. 249–265.

SILBERSCHATZ, A. "On the input/output mechanism in Concurrent Pascal." In *Proc. COMPSAC '77—IEEE Computer Society Computer Software and Applications Conference* (Chicago, Ill., Nov. 1977). IEEE, New York, 1977, pp. 514–518.

SILBERSCHATZ, A. "Communication and synchronization in distributed programs." *IEEE Trans. Softw. Eng.* **SE-5**, 6 (Nov. 1979), 542–546.

SILBERSCHATZ, A., KIEBURTZ, R. B., AND BERNSTEIN, A. J. "Extending Concurrent Pascal to allow dynamic resource management." *IEEE Trans. Softw. Eng.* **SE-3**, 3 (May 1977), 210–217.

SOLOMON, M. H., AND FINKEL, R. A. "The Roscoe distributed operating system." In *Proc. 7th Symp. Operating System Principles* (Pacific Grove, Calif., Dec. 10–12, 1979). ACM, New York, 1979, pp. 108–114.

SOUNDARARAJAN, N. "Axiomatic semantics of communicating sequential processes." Tech. Rep., Dep. of Computer and Information Science, Ohio State Univ., 1981.

SPECTOR, A. Z. "Performing remote operations efficiently on a local computer network." *Commun. ACM* **25**, 4 (Apr. 1982), 246–260.

U.S. DEPARTMENT OF DEFENSE. *Programming Language Ada: Reference Manual,* vol. 106, *Lecture Notes in Computer Science.* Springer-Verlag, New York, 1981.

VAN DE SNEPSCHEUT, J. L. A. "Synchronous com-

munication between synchronization components." *Inform. Process. Lett.* **13**, 3 (Dec. 1981), 127-130.

VAN WIJNGAARDEN, A., MAILLOUX, B. J., PECK, J. L., KOSTER, C. H. A., SINTZOFF, M., LINDSEY, C. H., MEERTENS, L. G. L. T., AND FISKER, R. G. "Revised report on the algorithm language ALGOL68." *Acta Inform.* **5**, 1-3 (1975), 1-236.

WEGNER, P., AND SMOLKA, S. A. "Processes, tasks and monitors: A comparative study of concurrent programming primitives." *IEEE Trans. Softw. Eng.*, to appear, 1983.

WELSH, J., AND BUSTARD, D. W. "Pascal-Plus—Another language for modular multiprogramming." *Softw. Pract. Exper.* **9** (1979), 947-957.

WELSH, J., AND LISTER, A. "A comparative study of task communication in Ada." *Softw. Pract. Exper.* **11** (1981), 257-290.

WETTSTEIN, H. "The problem of nested monitor cells revisited." *Oper. Syst. Rev.* **12**, 1 (Jan. 1978), 19-23.

WIRTH, N. "Modula: A language for modular multiprogramming." *Softw. Pract. Exper.* **7** (1977), 3-35. (a)

WIRTH, N. "The use of Modula." *Softw. Pract. Exper.* **7** (1977), 37-65. (b)

WIRTH, N. "Design and implementation of Modula." *Softw. Pract. Exper,* **7** (1977), 67-84. (c)

WIRTH, N. "Toward a discipline of real-time programming." *Commun. ACM* **20**, 8 (Aug. 1977), 577-583. (d)

WIRTH, N. *Programming in Modula-2.* Springer-Verlag, New York, 1982.

WULF, W. A., RUSSELL, D. B., AND HABERMANN, A. N. BLISS: A language for systems programming. *Commun. ACM* **14**, 12 (Dec. 1971), 780-790.

Received September 1982; final revision accepted February 1983