CONCEPTS AND NOTATIONS FOR CONCURRENT PROGRAMMING*

Gregory R. Andrews[+]
Fred B. Schneider[++]

TR 82-520
September 1982

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

[+]Department of Computer Science, University of Arizona, Tucson AZ 85721

[++]Department of Computer Science, Cornell University, Ithaca NY 14853

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

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

September 8, 1982

ABSTRACT

Much has been learned in the last decade about concurrent programming. This paper identifies the major concepts and describes some of the more important language notations for writing concurrent programs. The roles of processes, communication and synchronization are discussed from both an operational and an axiomatic viewpoint. 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

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

1. 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 can be used to express concurrent computations simply, that make synchronization requirements explicit, and that facilitate formal correctness proofs. Secondly, the availability of inexpensive processors has made possible the construction of distributed systems and multiprocessors that were previously economically infeasible. Concurrent programming is no longer the sole province of those who design and implement operating systems; it has become important in programming all kinds of applications, including database management systems, large-scale, parallel scientific computations and real-time, embedded control systems. In light of this, 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. This will require detailed discussions of some concurrent programming languages. However, we restrict attention to those whose designs we believe to be influential or conceptually innovative. Not all these languages 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 non-concurrency (but nevertheless important) issues.

We proceed as follows. In Section 2 we discuss the issues that underly all concurrent programming notations: how to express concurrent execution, how processes communicate, and how they synchronize. These issues are treated in detail in the remainder of the paper. In Section 3 we take a closer look at various ways to specify concurrent execution: coroutines, **fork** and **cobegin** statements, and **process** declarations. In Section 4 we discuss synchronization primitives for use when communication is performed using shared variables. Two general types of synchronization are considered—exclusion and condition synchronization—and a variety of mechanisms to implement them are described: busy-waiting, semaphores, conditional critical regions, monitors, and path expressions. In Section 5 we discuss message-passing primitives. Methods for specifying channels of communication and synchronization, and higher-level constructs for remote procedure call and atomic transactions are described. In Section 6 we identify three general classes of concurrent programming languages and compare them. Finally, in Section 7 we summarize the major topics and identify directions in which the field is headed.

2. Concurrent Programs: Processes and Process Interaction

2.1. 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*. Formulating a computation in such a manner is often a useful way to structure a program. For example, an airline reservation system that involves processing transactions from many terminals has a natural specification as a concurrent program: each terminal is controlled by a sequential process. Even when processes will not execute 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 and performs the specified computation, perhaps generating line images, which are stored in an output buffer. The *printer* process retrieves line images from the output buffer and writes them to a printer.

Execution of a concurrent program can be accomplished by running each process on its own processor (as would be possible in a distributed system or on a multiprocessor) [Jone80] or by allowing processes to share one or more processors. The first approach is referred to as *multiprocessing* or *parallel processing*. The latter approach is referred to as *multiprogramming*; it is supported by an operating system kernel [Dijk68a] that multiplexes the processes on the processor(s).

The rate at which processes are executed depends on which approach is used. When each process is executed on its own processor, it executes at a fixed, but perhaps unknown, rate; when processes share a processor, it is as if each is executing 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 how they interact, without regard for how they are executed, it is usual to 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 independent of how that program is executed, be it on multiple processors or on a single, multiprogrammed processor.

2.2. 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 communications can be 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. Recall that processes are executed with unpredictable speeds. Yet, to communicate, one process must perform some action that is detected by the other—for example, setting the value of a variable or sending a message. This only works if the events "perform an action" and "detect the action" are constrained to happen in that order. Thus, one can view synchronization as a set of constraints on the ordering of events. A *synchronization mechanism* is employed to delay execution of a process in order to satisfy such constraints.

To make this a bit more concrete, consider the batch operating system described above. A buffer (shared variable) 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 buffer if the buffer is empty.

This view of synchronization follows from taking an *operational approach* to program semantics. Execution of a concurrent program results a sequence of *atomic actions*, each resulting from the execution of an indivisible operation.† This sequence

†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

will be some interleaving of the sequences of atomic actions generated by each of the component processes. Rarely do all execution interleavings result in acceptable program behavior, as is illustrated in the following. Suppose initially $x=0$, 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 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 to it, and (iii) store the result in x. Then, 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—i.e. by controlling the ordering of the events corresponding to the atomic actions. (Then 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* [Floy67, Hoar69, Dijk76], provides a second framework in which to view the role of synchronization. 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*[†] provides an informal way to present a program and its proof. It consists of the program interleaved with assertions so that for each statement S,

---

made serially. This is a reasonable assumption in light of the way memory is constructed. See [LamL80b] for a discussion of some of the implications of relaxing this assumption.

[†]Sometimes called an asserted program.

the triple formed from the assertion that precedes S in the proof outline, the statement S, and the assertion that follows S in the proof outline is a theorem. For a programming logic that is sound with respect to a reasonable operational model, the appearance of an assertion R in a program text signifies that in *any* execution of the program that is started in a state that satisfies the program's precondition, R is true at 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 it [Ashc75, Kell76, Owic76a, Owic76b, Lamp77, LamL80a, Lamp82]. This is called *non-interference* and is illustrated in the following proof outline of two concurrent processes *P1* and *P2*.

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

Execution of *S2* does not interfere with the proof of $\{ x>0 \}\, x := 16\, \{ x=16 \}$ because the precondition of *S2* ($x<0$) is never true when either the pre- or postcondition of *S1* is. Hence, *S2* will never be executed in those states.

Synchronization mechanisms control interference in two ways. First, they can delay execution of a process until a given condition (assertion) is true. Thus, the precondition of the statement following the synchronization mechanism is guaranteed to be true prior to its execution.[†] Secondly, a synchronization mechanism can be used to ensure that a block of statements is an indivisible operation. This eliminates the possibility of concurrently executing processes interfering with assertions that appear within the proof of that block.

Both views of programs—operational and axiomatic—are useful. The operational approach—viewing synchronization as an ordering of events—is well-suited for explaining how synchronization mechanisms work. For that reason, it is used rather extensively in this survey. It also constitutes the philosophical basis for a family of synchronization mechanisms called *path expressions* [Camp74], which are

[†] Provided the assertion is not interfered with.

described in Section 4.5.

Unfortunately, the operational approach doesn't 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 that are processed concurrently in a data base system [BerP81]—it has only limited utility when applied to more complex concurrent programs [Akko78, Bern78]. This is 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 does not have this difficulty. It is perhaps the most promising technique for providing a way to understand concurrent programs. Some familiarity with formal logic is required for its use, however, and this has slowed its acceptance.

To summarize, in this section we have described three issues underlying the design of a notation for expressing a concurrent computation:

(i) how to express concurrent execution, i.e. the notion of process;

(ii) selection of the mode of interprocess communication; and

(iii) choice of a synchronization mechanism.

Synchronization mechanisms can be variously viewed as constraining the ordering of events or as controlling interference.

3. Specifying Concurrent Execution

Various notations have been proposed for specifying concurrent execution. Early proposals are marred by a failure to separate process definition from process synchronization. Later proposals separate these distinct concepts and are characterized by syntactic restrictions that impose some semblance of structure on a concurrent program. This allows easy identification of program segments that will be executed concurrently. Consequently, such proposals are well-suited for use with the axiomatic approach—proof obligations for establishing non-interference are made clear from the structure of the program. Below, some representative constructs for expressing concurrent execution are described.

3.1. Coroutines

Coroutines allow transfer of control among a collection of routines that do not exhibit a hierarchical relationship with respect to transfer of control [Conw63a]. Statements to implement coroutines were included in discrete event simulation languages such as SIMULA I [Nyga78] and its successors, and systems implementation languages including BLISS [Wulf71] and most recently Modula-2 [Wirt80].

Transfer of control among coroutines is accomplished using the **resume** statement. Execution of **resume** transfers control to the named routine, saving enough state information so that control can return to the instruction following the **resume**. (Execution of the first **resume** of a routine transfers control to the beginning of the routine.) Control can subsequently be returned to the original routine by executing another **resume**. Thus, the **resume** statement serves as the only way to transfer control among coroutines, and each coroutine starts at the beginning of its statement list only once. This is in contrast to subroutines where control is transferred by both **call** and **return**, and each subroutine starts over at the beginning each time it is called.

A use of coroutines appears below. 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 the diagram indicate the transfers of control.

program *P*;	**coroutine** *A*;	**coroutine** *B*;
...	...	...
call *A*;	...	...
...	**resume** *B*;	...
end	...	...
	...	**resume** *A*;
	resume *B*	...
		return

Each coroutine can be viewed as implementing a process. Execution of **resume** causes process synchronization. When used with care, coroutines are an acceptable way to organize concurrent programs that share a single processor. In fact, multiprogramming can be implemented using them. However, coroutines are not well suited for true parallel processing because their semantics allow for only one routine to be executed at a time.

3.2. The fork Statement

The **fork** statement [Denn66, Conw63b], like a **call** or **resume**, specifies that execution of a designated routine should commence. However, the invoking routine and the invoked routine proceed concurrently.

A use of **fork** is illustrated below.

program *P1*;	**program** *P2*;
...	...
L: **fork** *P2*;	...
...	...

Execution of *P2* is initiated when statement *L* of *P1* is executed; *P2* is then executed concurrently with statements following *L* in *P1*.

Because **fork** statements can appear in conditionals and loops, a detailed understanding of program execution is necessary in order to understand which routines will be executed concurrently. Nevertheless, when used in a disciplined manner, **fork** is practical and powerful. The UNIX operating system [Ritc74] makes extensive use of variants of **fork**. Similar statements have also been included in PL/I and Mesa [Mitc79].

3.3. The cobegin Statement

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

cobegin $S_1 \,||\, S_2 \,||\, ... \,||\, S_n$ **coend**

causes concurrent execution of S_1, S_2, ... 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** terminates only when execution of all the S_i's have terminated.

Although **cobegin** is not as powerful as **fork**—there are computations that can be specified with the latter but not with the former—it is sufficient for specifying the concurrent computations that arise in practice. Furthermore, the syntax of the **cobegin** statement makes explicit which routines are executed concurrently, and it is a single-entry, single-exit control structure. This allows the state transformation

+First called **parbegin** in [Dijk68b].

implemented by a **cobegin** to be understood in isolation, and then used to understand the program in which it appears without concern for the details of how the transformation is implemented.

Variants of **cobegin** have been included in the concurrent programming languages Edison [Brin81] and Argus [Lisk82].

3.4. Process Declarations

Large programs are often structured as a collection of sequential routines, which are executed concurrently. Although such a computation could be specified using a **cobegin** statement, it seems desirable to provide a syntactic unit analogous to the module [Parn72] for this purpose. The *process declaration* provides such a facility. A collection of process declarations is equivalent to a single **cobegin** where each of the declared processes is a component statement (S_i above).

Use of process declarations to structure a concurrent program is illustrated in Figure 1, which is an outline for the batch operating system described earlier. We shall use this notation for process declarations in the remainder of this paper.

Facilities for process declarations appear in many contemporary concurrent programming languages, such as Concurrent Pascal [Brin75], Modula [Wirt77a], CSP [Hoar78], PLITS [Feld79], Ada [Refe80], and SR [Andr82].

4. 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 that are implemented by sequences of statements. If processes concurrently perform operations on the same shared data object, then unintended results might occur due to the interleaving of those statements. 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

```
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 1. Outline of batch operating system.

execution interleavings are visible only if two computations access shared variables. Then, execution of one can obtain results corresponding to incomplete execution of the other. Thus, if two routines have no variables in common then their execution need not be mutually exclusive.

Another situation where it is necessary to coordinate execution of concurrent processes occurs when a shared data object is not in a state conducive to the execution of a particular operation. A process attempting to perform such an operation should be delayed, since the state of the data object (i.e. the values of the variables that comprise the object) may subsequently change as a result of operations performed by other processes. We shall call this *condition synchronization*.† Examples of condition synchronization appear in the batch operating system discussed above:

†Unfortunately, there is no commonly agreed upon term for this type of synchronization.

for example, a process attempting to execute a deposit operation on a buffer—a shared data object—should be delayed if there is no space in the buffer. 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 types of synchronization.

4.1. Busy Waiting

One way that processes can synchronize is by testing and setting the values of shared variables. A variable is set by a process when an event of interest occurs, and processes delay their execution by repeatedly testing that variable until it is found to have a certain value. Because a process waiting for a variable to change value must repeatedly test this 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 illustrate this approach to implementing synchronization, we present a solution to the two-process mutual exclusion problem [Pete81]. (This protocol is simpler than the classic solution proposed by Dekker [Shaw74].) The solution involves an *entry protocol*, which is executed by a process before entering its critical section, and an *exit protocol*, which is executed by a process after finishing its critical section.

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

Three shared variables are used to realize the desired synchronization. Boolean variable *enteri* 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 critical section; it is used when both processes execute their respective entry protocols at about the same time. The entry and exit protocols for process *P1* are given below. The protocols for *P2* can be obtained by interchanging appearances of "*enter1*" and "*enter2*", and "*P1*" and "*P2*". Variables *enter1* and *enter2* are initially assumed to be false; *turn* may start as "*P1*" or "*P2*".

```
process P1;
  loop
    { Entry Protocol }
      enter1 := true;    { announce intent to enter }
      turn := "P2";   { set priority }
      { wait until other process not in or it is my turn }
        while enter2 and turn="P2" do skip;
    Critical Section;
    { Exit Protocol }
      enter1 := false;   { renounce intent to enter }
    Noncritical Section
    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 each waiting for an event that will never occur. Above, deadlock could occur if each process could spin forever in its entry protocol; the use of *turn* precludes this. The second desirable property is *fairness*: if a process is trying to enter its critical section, it will be able to do so after a finite time provided the other process exits its critical section after a finite time. This is a desirable property of a synchronization mechanism because then 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 with a bound of one; the use of *turn* also provides this. Pete81 contains operational proofs of mutual exclusion, deadlock freedom and fairness; Dijk81a contains axiomatic ones.

Synchronization protocols that just use busy-waiting are difficult to design, understand and prove correct. The existence of instructions that make two memory references as part of a single indivisible operation—e.g. the TS (test-and-set) instruction on the IBM 360/370 processors—helps, but does not significantly simplify, the task of designing synchronization protocols. Also, busy-waiting wastes processor cycles. A processor executing a spinning process can usually be employed more productively by running other processes until the awaited event occurs. Lastly, this 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 which program variables are used for implementing synchronization and which are used, say, for interprocess communication.

4.2. 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 [Dijk68a, Dijk68b]. A *semaphore* is a non-negative 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 an indivisible operation.[†] 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, a choice exists as to 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 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. Below, we show a solution to the two-process mutual exclusion problem in terms of semaphores.

[†] 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" [Dijk81b]. Reflecting on the definitions of **P** and **V**, Dijkstra and his group observed that 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**.

```
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.
```

Semaphores can also be used to solve selective mutual exclusion problems where shared variables have been partitioned into disjoint sets. In this case, a semaphore is associated with each set and used to control access to the variables in that set. Thus, critical sections that reference variables in the same set execute with mutual exclusion, but critical sections that reference variables in different sets can execute concurrently.

Semaphores can also be used to implement condition synchronization. In this case, a **V** operation signals the occurrence of an event, and a **P** operation delays process execution if an event of interest has not yet occurred.

These uses of semaphores are illustrated in an implementation of our simple operating system in Figure 2. 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*.[+] Thus, 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:

[+]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);
      num_cards, num_lines : semaphore initial (0);
      free_cards, free_lines : semaphore initial (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 2. Batch operating system with semaphores.

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 the same roles with respect to *output_buffer*.

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. To do this, it maintains a *ready list*—a queue of descriptors for processes that are eligible to be executed on the processor—and multiplexes the processor among these processes, running each process for some period of time. The 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, the semaphore is decremented if it will remain non-negative; otherwise the descriptor for the executing process is moved to the queue associated with the semaphore. For a **V** operation, if the queue associated with the semaphore is not empty, one descriptor is moved from it to the ready list; otherwise the semaphore is incremented.

This approach to implementing synchronization mechanisms is quite general and will be applicable to the other mechanisms we 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. If the ready list is shared, it is subject to concurrent access, which requires that mutual exclusion be ensured. Usually, busy-waiting protocols are used 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 another kernel.

4.3. Conditional Critical Regions

Although semaphores can be used to program just about any type of synchronization, **P** and **V** are rather unstructured primitives, and it is quite easy to make an error 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—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 for 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—one must look 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 [Hoar72, Brin72, Brin73b] overcomes these difficulties. It provides a structured notation for specifying synchronization. Shared variables are grouped into *resources* and declared as such. Consequently, a compiler can check that a shared variable is referenced only from within a conditional critical region associated with the corresponding resource. Code generated by the compiler guarantees that execution of the conditional critical regions associated with a given resource is mutually exclusive.

A resource *r* containing variables *v1*, *v2*, ... , *vn* is declared as follows:

resource *r* : *v1*, *v2*, ... , *vn*;

The variables in the resource are accessed by means of *conditional critical region* (CCR) statements of the form

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

where *B* is a Boolean expression and *S* is a statement list. *B* may reference only variables in *r* and variables that are local to the process in which this statement appears. Furthermore, a variable can be in at most one resource and variables in resource *r* can be accessed only in CCR statements that name *r*.

We say that a process is *active* in a conditional critical region associated with r if that process is evaluating B or executing S in a CCR statement where r is the named resource. Execution of a CCR statement in process P is as follows. If no other process is active in a conditional critical region associated with r and B is true, then S is executed immediately. Execution of other conditional critical regions associated with r is never interleaved with evaluation of B or execution of S. Hence, B is guaranteed to be true when execution of S is commenced and S executes as an indivisible operation with respect to other CCR statements that name r. On the other hand, if B is not true or if a process is already active in a conditional critical region associated with r, P is delayed. The delay mechanism is assumed to be fair in the sense that a process awaiting a condition B that is true infinitely often will eventually be allowed to continue.

Note that this synchronization mechanism separates mutual exclusion from condition synchronization. Condition synchronization is made explicit in B; mutual exclusion is implicit. CCR statements can be expensive to implement, however. Because B can contain references to local variables, each process must evaluate its own conditions. On a multiprogrammed processor, this results in numerous context switches—i.e. frequent saving and restoring of process states. If each process is executed on its own processor and memory is shared, however, CCR statements can be implemented quite cheaply using busy-waiting. CCR statements are the synchronization mechanism in the Edison language [Brin81], which is designed specifically for multiprocessor systems.

Figure 3 contains our batch operating system example, programmed using conditional critical regions.

Programs written in terms of conditional critical regions can be understood quite simply using the axiomatic approach. Each CCR statement implements an operation on a shared object and each resource r (shared object) is characterized by an *invariant relation* I_r—a predicate that is true of its valid states. For example, in *OPSYS* the operations are ones to insert and remove items from bounded buffers and the buffers *inp_buff* and *out_buff* both satisfy the invariant:

IB: $0 \le head, tail \le N-1$ **and**
$0 \le size \le N$ **and**
$tail = (head + size)$ **mod** N **and**
$slots[head], \ldots, slots[tail-1]$ contain the *size* most
recently inserted items in oldest to youngest order

The Boolean expression B in a conditional critical region 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 exclude each other in time. Hence, the proofs of processes are interference-free as long as variables local to a process appear only in the proof of that process. Thus, a concurrent program can be understood in terms of its component sequential processes: the interactions due to concurrent execution are subsumed by the invariant and the B's.

4.4. Monitors

In addition to being costly to implement on single processors, conditional critical regions have the attribute that 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 a resource definition and operations that manipulate it [Dijk68b, Brin73a, Hoar74]. This allows a resource that is subject to concurrent access to be viewed as a module [Parn72]. 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.

4.4.1. Definition

A monitor consists of a collection of *permanent variables*, which are used to store the state of the resource, and some procedures, which implement operations on the resource. The values of the permanent variables are retained between activations of monitor procedures and may be accessed only from within the monitor. A monitor also has an initialization section to allow the values of the permanent variables to

```
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 3. Batch operating system with CCR statements.

be set before any of the monitor procedures is invoked. Monitor procedures can have parameters and local variables, which are allocated only for the duration of the procedure invocation. The structure of a monitor with name *mname* and procedures *op1*, ... , *opN* is shown in Figure 4.

```
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 4. Monitor structure.

A monitor procedure is invoked by executing

call *mname.opj*(arguments)

where *mname* is the name of a monitor and *opj* is the name of a procedure defined by monitor *mname*. 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 [Hoar74] 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 *cond* is a conditional 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.

As an example of a monitor, Figure 5 defines a bounded buffer type. Our batch operating system can be programmed using three processes that communicate using two instances of this type, as shown in Figure 6.

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

[+]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.

```
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 5. Bounded buffer monitor.

cond.**wait**(p)

has the same semantics as *cond*.**wait**, except processes blocked on condition variable *cond* are awakened in ascending order of p.‡

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: *request*(*time* : *integer*), which is called by users to request access to the resource for *time* units, and *release*, which is called when a user has finished using the resource.

‡Consequently, condition variables used in this way are not necessarily fair.

```
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 6. Batch operating system with monitors.

```
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
```

4.4.2. Other Approaches to Condition Synchronization

4.4.2.1. Queues and Delay/Continue

In Concurrent Pascal [Brin75], 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 defining an array of queues and performing a **continue** operation on the 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. As before, a process blocked on the selected queue resumes execution of the monitor procedure in which it was delayed.

It is easier to implement **continue** than **signal** because **signal** requires code to ensure that processes suspended by **signal** operations re-acquire control of the monitor before processes attempting to begin execution in that monitor. In both cases the objective is to ensure that a condition is not invalidated between the time it is signaled and the awakened process actually resumes execution. While **continue** is cheaper to implement than **signal**, it is less powerful. A monitor written using condition variables cannot always be translated directly into one that uses queues without also adding monitor procedures [Howa76a]. 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.

4.4.2.2. Conditional Wait and Automatic Signal

In contrast to semaphores, **signals** on condition variables are not saved—a process always delays as a result of executing **wait**, even if a previous **signal** did not awaken any process.† This can make **signal** and **wait** difficult to use correctly, because other variables must be used to record that a **signal** was executed. These

†Our arguments about condition variables also apply to queues.

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 each **wait** on a condition variable is contained in an **if** statement in which the Boolean expression is the negation of the desired condition synchronization, and if 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 signaling process resumes execution.

To mitigate these difficulties, Hoare 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) causes the invoker to be delayed 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 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 conditional wait was proposed by Kessels [Kess77] for use when only permanent variables appear in B. The buffer monitor above satisfies this requirement. 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 Boolean expressions associated with all conditions on which there are waiting processes are evaluated. If one is found to be true then one of the waiting processes is granted control of the monitor. If none is found to be true then a new activation of one of the monitor's procedures is permitted.

Using Kessels' proposal, the buffer monitor given earlier 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

Secondly, 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, conditions being awaited are explicitly specified. The primary limitation of the proposal is that it cannot be used to solve most scheduling problems, because operation parameters, which are local variables, may not appear in conditions.

4.4.2.3. Signals as Hints

A different approach to condition synchronization was employed when monitors were added to Mesa [LamB80]. 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.[†] 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 was the case for the conditional wait and 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 timeout 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 to perform another **wait** or 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 is useful if more than one process could proceed when a condition becomes true. It 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 [Ritc74].

[†]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.

4.4.3. 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 as long as no process is executing in the monitor. Thus, the monitor invariant must be reestablished by a process before exiting the monitor or performing a **wait** (**delay**) or **signal** (**continue**). Then the monitor invariant can be assumed true of the permanent variables whenever a process acquires control of the monitor—whether control is acquired by calling a monitor procedure or upon reactivation following a **wait** or **signal**.

The fact that monitor procedures are mutually exclusive simplifies non-interference proofs. One need not consider interleaved execution of monitor procedures. However, interference can arise when programming condition synchronization.

A process may need to delay its progress in order to implement medium-term scheduling, or because although the monitor invariant is true, some other condition must be true for execution of the invoked procedure to continue. (The invariant will probably not be so strong that it ensures that any procedure can be executed at any time.) Mechanisms that delay a process cause its execution to be suspended and control of the monitor to be relinquished with the understanding that some condition B in addition to the monitor invariant will be true when the process is next executed. The truth of B when the process awakens can be ensured by checking for it automatically or by requiring the programmer to build these tests into the program. If programmed checks are used, they can appear in the process that establishes the condition (for condition variables and queues) or in the process that performed the **wait** (the Mesa model). However, if the signaler checks for the condition, we must ensure that the condition is not destroyed between the time the **signal** occurs and when the blocked process actually executes. That is, we must ensure that the condition is not interfered with by other execution in the monitor. Interference might be caused by the process that established the condition in the first place—if the signaler does not immediately relinquish control of the monitor (i.e. if **notify** is used)—or by some process that executes after the condition has been signaled—if the signaled process does not get reactivated before new calls of monitor procedures are allowed to execute (which can happen in Modula [Wirt77a]). Proof rules for monitors and the various signaling disciplines are discussed in [Howa76a, Howa76b].

4.4.4. 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 been the cause of much discussion [List77, Hadd77, SIGO78]. The controversy concerns 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, this will not be the case in monitors from which nested calls had been made. Processes that attempt to invoke procedures in these monitors will become blocked, and so on. This has performance implications since the amount of concurrency exhibited by the system will be decreased.

The nested monitor call problem can be attacked in a number of ways. One approach is to prohibit nested monitor calls, which was done in SIMONE [Kaub76] and Modula [Wirt77a] (for monitors that are not lexically nested). 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. This would require that the monitor invariant be established before any monitor call that will cause the process to block. Since it cannot be known *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 [Silb77] for handling dynamic resource allocation problems and the *scheduler monitor* [Schn78] for scheduling access to shared resources are based on this line of thought.

The last approach to the nested monitor call problem, and probably the most sane, is to appreciate that monitors are only a structuring tool for resources that are subject to concurrent access [Andr77, Parn78]. 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 where the operations provided by a given monitor can be executed concurrently without adverse effects, and even cases where more than one instance of the same monitor procedure can be executed in parallel (e.g. a read procedure in a monitor that encapsulates a database). Monitor procedures can be executed concurrently provided they do not interfere with each other. Also, there are cases where 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, Andr77 defines a monitor-like construct that allows the programmer to specify that certain monitor procedures be executed concurrently and that mutual exclusion be released for certain calls.

4.4.5. 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 [Mitc79] and Pascal-Plus [Wels79].

4.4.5.1. Concurrent Pascal

Concurrent Pascal [Brin75, Brin77] 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 [Brin76a, Brin76b]; Job Stream, a batch operating system for processing Pascal programs; and a real time process control system [Brin77].

One of the major goals of Concurrent Pascal was to ensure that programs exhibited reproducible behavior [Brin77]. Monitors ensured that pathological interleavings of concurrently executed routines that shared data were no longer possible—the compiler generated 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 their declaration.

†A compiler that generated code for the PDP 11/45 was available for use in May 1975 [Hart77].

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 away the notion of an interrupt.

Various aspects of Concurrent Pascal, including its approach to I/O, are analyzed in [Loeh77, Silb77, Keed79].

4.4.5.2. Modula

Modula was developed for programming small, dedicated computer systems, including process control applications [Wirt77a, Wirt77b, Wirt77c, Wirt77d]. 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 [Wirt77c]. It does not timeslice the processor among processes, as is done in Concurrent Pascal. 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 restrictions associated with them 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 in [BerA81].

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, in Modula support for devices is 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 an interrupt is received from the corresponding device. Thus, interrupts are viewed as **signal** operations (**send** in Modula) 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; this allows 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. But, 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 are evaluated in [Andr79, Hold80, BerA81].

4.5. 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 equivalent mechanism). Consequently, synchronization of operations on the resource implemented by a monitor is realized by code that is scattered throughout the monitor. Some of this code is visible to the programmer—e.g. **signal** and **wait**. Other code, such as the code that ensures mutual exclusion of monitor procedures, is not.

Another approach to defining a shared resource 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 [Camp74]. Subsequently, extensions and variations have been proposed [Habe75, Laue75, Camp76, Flon76, Laue78, Andl79]. Below, we describe one specific proposal [Camp76] that has been incorporated into Path Pascal, an implemented systems programming language [Camp79].

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. One or more 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 executions of *path_list*, and "[*path_list*]" to specify an unbounded number of concurrent executions of *path_list*.

For example, the path expression

path *deposit, fetch* **end**

specifies that operations *deposit* and *fetch* can execute concurrently, while

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

specifies that *deposit* and *fetch* are mutually exclusive and can be executed in any order. A module implementing a bounded buffer of size one might well contain the path

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

to specify that the first invoked operation be a *deposit*, and that each *deposit* be followed by a *fetch*. Synchronization constraints for a bounded buffer of size N are specified by:

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

This ensures that (i) executions of *deposit* are mutually exclusive, (ii) executions of *fetch* are mutually exclusive, (iii) execution of each *fetch* is preceded by an execution of *deposit* and (iv) at most N *deposit* operations have not been followed by *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 previously. For this reason, there is no variable *size* because it would have been subject to concurrent access.

As a last example,

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

specifies that either an unbounded number of concurrent *reads* or a single *write* may be executed simultaneously. This specifies the "weak reader's preference" solution to the readers/writers problem [Cour71].

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, where each sentence in the language 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. Whether an operation can be executed might depend on the state of a resource in a way that is 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 Sec. 4.4.1 is an example of a resource where the value of a parameter 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 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. A 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.

5. 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 so that they convey data as well as implementing 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 upon receiving a message a process 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* permits control over the source of the message, i.e. which **send** statements could have sent it. Receipt of a message causes assignment of the values in the message to the variables in *variable_list*.

Designing message-passing primitives involves making choices about the form and semantics of these general commands. Two main issues must be addressed: how source and destination designators are specified and how communication is synchronized. Common alternatives 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.

5.1. 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 of a message that was sent by the *executer* process.

Direct naming is easy to implement and use. It makes it possible for a process to control when it receives messages from each other process. Our simple batch operating system might be programmed using direct naming as shown in Figure 7.

```
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 7. Batch operating system with message passing.

The batch operating system also illustrates an important paradigm for process 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—e.g., the lineprinter driver and the *printer* process in our operating system example—is an illustration of this paradigm. The lineprinter driver is a server; it repeatedly receives a request 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 fine; difficulties arise if there is more than one client—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 result in 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 **send** statements in any process and as the source designator in **receive** statements in any process. Thus, messages sent to a given mailbox might 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 [Gele82]. 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 where a mailbox name can appear as the source designator in **receive** statements in only one process does not suffer these implementation difficulties. Such mailboxes are often called *ports* [Balz71]. 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 single-server/multiple-clients problem. (The multiple-server/multiple-clients problem 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 the time the program is compiled. This limits the ability of a program to exist in a changing environment. For example, this would preclude implementing UNIX filters [Ritc74]. The second problem is that if a program might *ever* need access to a channel, it must permanently have that access. In many applications, such as file systems, it is 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 and destination designators. These variables can be viewed as containing *capabilities* for the communications channel [Bask77, Solo79, Andr82].

5.2. Synchronization

Another important property of message-passing statements concerns whether their execution could cause a delay. A statement is *non-blocking* if its execution never causes its invoker to be delayed; 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 the message could not be sent because the buffer is full. Similarly, execution of a **receive** when no message that satisfies the source designator is available for receipt might cause a delay or might terminate with a code signifying that no message was available.

If there is 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 ahead of a receiver. Consequently, when a message is received, it contains information about a state of the sender that is not necessarily its current state. At the other extreme, with no buffering execution of a **send** is always delayed until a corresponding[+] **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 correspond to the current state of the sender. 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.

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 do provide a non-blocking **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 (e.g. to schedule them).

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

Sometimes, further control over the messages that 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 delivered messages before receiving one.

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 non-blocking **receive**, busy-waiting is required. However, blocking message-passing statements can achieve the same semantic effects as non-blocking ones by using *selective communications*,[‡] which is based on guarded commands [Dijk75].

A *guarded command* has the form

guard → *statement*

The guard consists of a Boolean expression, optionally followed by a message-passing statement. A guard *succeeds* if the Boolean expression is true and the message-passing statement would not cause a delay; a guard *fails* if the Boolean expression is false. The alternative statement

if *G1* → *S1*
□ *G2* → *S2*
...
□ *Gn* → *Sn*
fi

is executed as follows. All the guards are evaluated. One of those that succeeds is selected non-deterministically, the corresponding message-passing statement is executed (if present), and then the following guarded statement is executed. If all guards fail then the command aborts. Otherwise, execution is delayed until either all guards fail or some guard succeeds. 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.

‡There is no commonly agreed upon term for this.

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

```
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

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

Implementing selective communications when **receive** statements can appear in guards but **send** statements cannot is reasonably straightforward. Development of efficient protocols to support selective communications when both **send** and **receive** statements can appear in guards remains an active research area [Schw78, Silb79, Bern80, Vand81, Schn82, Reif82].

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

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 artificial; a producer/consumer relationship is inherently symmetric and the program should mirror this fact.

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 Sec. 4.4.1) 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*

5.3. Higher-Level Message-Passing Constructs

5.3.1. 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**; the server a **receive** followed by a **send**. Because this type of interaction is very common, higher-level statements that support it directly have been proposed. These are termed *remote procedure call* statements because of the interface 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 it designates the server process, if port or mailbox naming is used it 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 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 request a service but forget to wait for the results.

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:+

+This is another reason this kind of interaction is termed remote procedure call.

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 [Cook80, Andr82], in which case **call**s to the same remote procedure would execute sequentially. Alternatively, a new process can be created for each execution of **call** [Brin78, Lisk82]—these could execute concurrently which means that the different instances of the server might need to synchronize if they share variables.

In the second approach to specifying the server side of a remote procedure call, 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.

Different semantics result depending on whether the reply message is sent by a synchronous or 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. Use of asynchronous **send** does not allow this, but does not delay the server, either.

In the second approach to specifying the server side, remote procedure call is called a *rendezvous* [Refe80] 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 this approach is that client **calls** may be serviced at times of the server's choosing—e.g. **accept** statements 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 enable a server to wait for and select one of several requests to service [Refe80, Andr81]. 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. This is 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 using it. For example, the *shortest_next_allocator* of the previous section still requires two client/server exchanges to process allocation requests. This is 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 small number of different scheduling priorities, this can be overcome by associating a different server operation with each priority level. Ada [Refe80] 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 [Andr81] includes such a mechanism (see Sec. 5.5.4).

5.3.2. Atomic Transactions

An often-cited advantage of distributed 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.[†]

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 [Nels81, Spec82]. 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 timeout interval to the remote **call**; if no response is received by the client before the timeout interval expires then the client presumes the server has failed and takes some action.

[†]For a general discussion, the interested reader is referred to [Kohl81].

Deciding what action to take after a failure has been detected can be difficult. In case (i) above, the correct action would be to retransmit the message. In case (ii), however, that action would cause a second execution of the remote procedure body. This is an undesirable state of affairs unless the procedure is *idempotent*—i.e. repeated execution has the same effect as a single execution. Finally, the correct action in case (iii) depends 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 can be handled by saving state information, called *checkpoints*, and programming special recovery actions. A more general solution is to view execution of a remote procedure in terms of atomic transactions.

An *atomic transaction* [Reed79, Lamp81] 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*, 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 [Lisk82, Spec82]. This is done by making execution of a remote procedure an atomic transaction that is allowed to "commit" only after the reply has been 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 [Lisk82].

The main consideration in the design of these mechanisms is that it not be possible for a process to see system data in an inconsistent state resulting from partial execution of a remote procedure. The use of atomic transactions is one way to do this, but it is quite expensive [Lamp79, Lisk81]. Other techniques to ensure invisibility of inconsistent states have been proposed [Lync81, Schl81] and this remains an active area of research.

5.4. An Axiomatic View of Message Passing

A primary motivation for the development of structured primitives for controlling process interaction when shared variables are used was to restrict execution interleavings and thereby eliminate interference. 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 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 [Schl82a]. 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 the message was sent; this 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 additional work—i.e. define more processes—to allow a high degree of parallel activity or will have to do additional work—i.e. use 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 "acknowledgement" protocols, for example. These schemes are studied in [Schl82b], 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 [Good79]. Proof systems for synchronous message-passing statements—in particular the input and output commands in CSP—are described in [Apt80, Levi81, Misr81, Lamp82, Schl82a]. Proof rules for asynchronous message-passing and remote procedures are described in [Schl82a].

5.5. 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, as the two major message-passing design issues—channel naming and synchronization—are orthogonal so 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.

5.5.1. Communicating Sequential Processes

Communicating Sequential Processes (CSP) [Hoar78] 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 are executing a matching pair of input/output commands. The result of communication is that the value of the expression 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 Sec. 5.2.

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 in [Hoar78] and is described in detail in [Kieb79]. Recently, languages based on CSP have also been described [Jaza80, Mao80, Rope81].

5.5.2. PLITS

PLITS, an acronym for "Programming Language In The Sky," was developed at the University of Rochester [Feld79]. 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 [Ball76].

A PLITS program consists of a number of modules; *active modules* are processes. Message passing is the sole means for inter-module 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. To enable a process to avoid blocking when there is no message available for receipt, PLITS also provides primitives to test whether messages with certain field values or transaction keys are available for receipt.

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 inter-processor communication.

5.5.3. Ada

Ada [Refe80] is a DoD sponsored 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 5.3.1. 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 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 [Hoar81], much of which is not related to concurrency. But, few applications using the concurrent programming features have been programmed, and, at the time of this writing, no compiler for full Ada has been completed. Implementation of some of the concurrent programming aspects of Ada is likely to be hard. A paper by Welsh and Lister [Wels81] compares the concurrency aspects of Ada to CSP and Distributed Processes [Brin78].

5.5.4. SR

SR (Synchronizing Resources) [Andr81, Andr82], 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*. 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 capability 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 there is more than one pending invocation 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 re-evaluation 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 should usually be 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.

5.5.5. Some Other Language Notations Based on Message Passing

Gypsy [Good79], one of the first high-level languages based on message passing, uses mailbox naming and buffered message-passing. A major focus of Gypsy was development of a programming language well suited for the construction of verifiable systems. It has been used to implement special-purpose systems for single- and multi-processor architectures.

Distributed Processes (DP) [Brin78] was the first language to be based on remote procedure calls. In DP, remote procedures are specified as externally callable procedures within a server process. When called, they are executed as soon as the server is idle. Thus, at any time the server process either is executing its main body, is executing the body of one its remote procedures, or is idle. Internal synchronization is specified by a variant of conditional critical regions.

StarMod [Cook80] synthesizes aspects of Modula and Distributed Processes. It borrows modularization ideas from Modula and communication ideas from Distributed Processes. It retains the Distributed Processes concept of procedure-like ports in a receiving process—implemented by single processes, however—but also includes the option of servicing ports by executable statements. 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. Synchronization within a module is provided by semaphores.

Argus [Lisk82] also borrows ideas from Distributed Processes—remote procedures implemented by dynamically created processes that 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.

6. 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-oriented languages provide message passing (i.e. **send** and **receive**) as the primary means for process interaction. In contrast to procedure-oriented languages, there are no shared, passive objects, 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 [Habe80]; 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 [Nels81, Spec82].

In a recent paper, Lauer and Needham argued that procedure-oriented and message-oriented languages are duals in terms of expressive power, logical equivalence, and performance [Laue79]. (They did not consider operation-oriented languages, which have only recently come into existence.) Their thesis was examined in depth by Reid [Reid80], 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 uses of these mechanisms is a bad idea. We, however, favor programming in the style appropriate to the language.

7. 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 resulted in the development of synchronization primitives, a number of which are described in this paper. The historical and conceptual relationships between these primitives is illustrated in Figure 8.

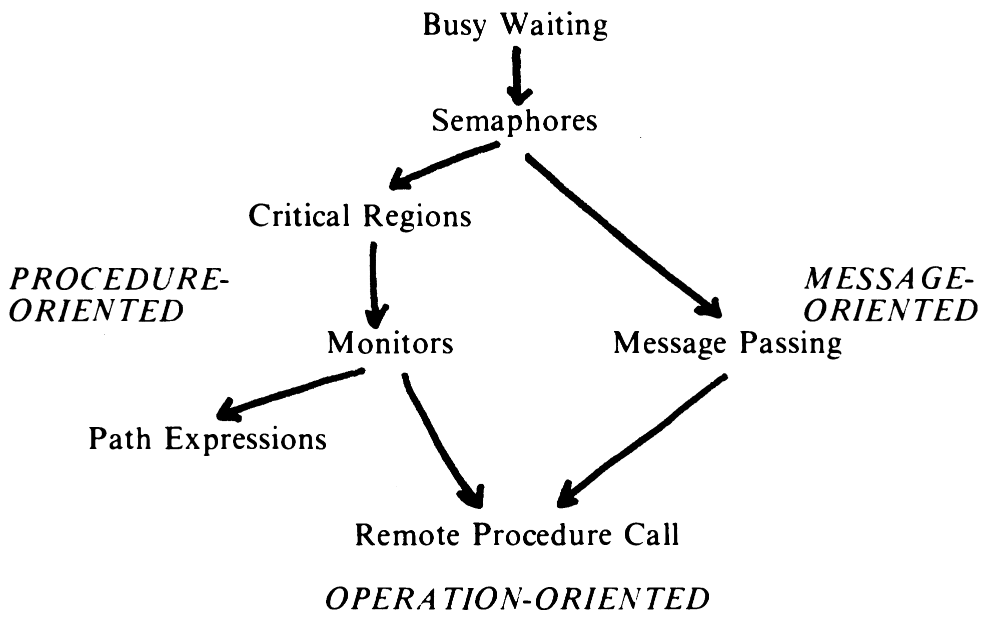

Figure 8. Synchronization techniques and language classes.

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: constructs were defined that enforced their structured use, resulting in critical regions, monitors and path expressions; and "data" was added to the synchronization associated with semaphores, resulting in message-passing primitives. Finally, these two divergent approaches were unified with the definition of remote procedure calls.

Since the first concurrent programming languages were defined only a decade ago, practical experience has increased our understanding 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 8.

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 towards 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 in not well understood. Little is known about the design of distributed (decentralized) concurrent programs. Lastly, 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 (the many) earlier drafts of this survey: David Gries, Phil Kaslo, Lynn Kivell, Gary Levin, Ron Olsson, Rick Schlichting, and David Wright.

References

[Akko78] Akkoyunlu, E.A., Bernstein, A.J., Schneider, F.B., and Silberschatz, A. Conditions for the equivalence of synchronous and asynchronous systems. *IEEE Trans. on Software Engineering SE-4*, 6 (Nov. 1978), 507-516.

[Andl79] Andler, S. Predicate path expressions. *Proc. 6th ACM Symposium on Principles of Programming Languages*, San Antonio, Jan. 1979, 226-236.

[Andr77] Andrews, G.R., and McGraw, J.R. Language features for process interaction. *Proc. ACM Conference on Language Design for Reliable Software, SIGPLAN Notices 12*, 3 (March 1977), 114-127.

[Andr79] Andrews, G.R. The design of a message switching system: an application and evaluation of Modula. *IEEE Trans. on Software Engineering SE-5*, 2 (March 1979), 138-147.

[Andr81] Andrews, G.R. Synchronizing resources. *ACM Trans. on Prog. Languages and Systems 3*, 4 (Oct. 1981), 405-430.

[Andr82] Andrews, G.R. The distributed programming language SR—mechanisms, design, and implementation. *Software—Practice and Experience 12*, 8 (Aug. 1982). 719-754.

[Apt80] Apt, K.R., Francez, N., and de Roever, W.P. A proof system for communicating sequential processes. *ACM Trans. on Prog. Languages and Systems 2*, 3 (July 1980), 359-385.

[Ashc75] Aschcroft, E.A. Proving assertions about parallel programs. *Journal Computer Systems 10* (Jan. 1975), 110-135.

[Ball76] Ball, E., et al. RIG, Rochester's intelligent gateway: System overview. *IEEE Trans. on Software Engineering SE-2*, 4 (Dec. 1976), 321-328.

[Balz71] Balzer, R.M. PORTS—a method for dynamic interprogram communication and job control. *Proc. AFIPS SJCC Computer Conf.*, Vol. 39 (1971). 485-489.

[Bask77] Baskett, F., Howard, J.H., and Montague, J.T. Task communication in DEMOS. *Proc. Sixth Symp. on Operating Systems Principles*. West Lafayette, Indiana, Nov. 1977, 23-31.

[Bern78] Bernstein, A.J., and Schneider, F.B. On language restrictions to ensure deterministic behavior in concurrent systems. *Proc. 3rd Jerusalem Conference on Information Technology JCIT3*, J. Moneta (ed.), North Holland Publishing Co., 1978, 537-541.

[Bern80] Bernstein, A.J. Output guards and nondeterminism in "communicating sequential processes". *ACM Trans. on Prog. Languages and Systems 2*, 2 (April 1980), 234-238.

[BerA81] Bernstein, A.J., and Ensor, J.R. A modification of Modula. *Software—Practice and Experience 11* (1981), 237-255.

[BerP81] Bernstein, P., and Goodman, N. Concurrency control in database systems. *ACM Computing Surveys 13*, 2 (June 1981), 185-221.

[Brin72] Brinch Hansen, P. Structured multiprogramming. *Comm. ACM 15*, 7 (July 1972), 574-578.

[Brin73a] Brinch Hansen, P. *Operating System Principles*. Prentice-Hall, New Jersey, 1973.

[Brin73b] Brinch Hansen, P. Concurrent programming concepts. *ACM Computing Surveys 5*, 4 (Dec. 1973), 223-245.

[Brin75] Brinch Hansen, P. The programming language Concurrent Pascal. *IEEE Trans. on Software Engineering SE-1*, 2 (June 1975), 199-206.

[Brin76a] Brinch Hansen, P. The Solo operating system: job interface. *Software—Practice and Experience 6* (1976), 151-164.

[Brin76b] Brinch Hansen, P. The Solo operating system: processes, monitors, and classes. *Software—Practice and Experience 6* (1976), 165-200.

[Brin77] Brinch Hansen, P. *The Architecture of Concurrent Programs*. Prentice-Hall, New Jersey, 1977.

[Brin78] Brinch Hansen, P. Distributed processes: A concurrent programming concept. *Comm. ACM 21*, 11 (Nov. 1978), 934-941.

[Brin81] Brinch Hansen, P. Edison: a multiprocessor language. *Software—Practice and Experience 11*, 4 (April 1981), 325-361.

[Camp74] Campbell, R.H., and Habermann, A.N. The specification of process synchronization by path expressions. *Lecture Notes in Computer Science 16*, Springer-Verlag, Heidelberg (1974), 89-102.

[Camp76] Campbell, R.H. Path expressions: a technique for specifying process synchronization. Ph.D. Dissertation, Computing Laboratory, University of Newcastle upon Tyne, August, 1976.

[Camp79] Campbell, R.H., and Kolstad, R.B. Path expressions in Pascal. Dept. of Computer Science, University of Illinois at Urbana, Urbana, Illinois, January, 1979.

[Conw63a] Conway, M.E. Design of a separable transition-diagram compiler. *Comm. ACM 6*, 7 (July 1963), 396-408.

[Conw63b] Conway, M.E. A multiprocessor system design. *AFIPS Conf.* Proc. Vol. 24, Spartan Books, Baltimore, 1963, 139-146.

[Cook80] Cook, R.P. *MOD—A language for distributed programming. *IEEE Trans. on Soft. Engr. SE-6*, 6 (Nov. 1980), 563-571.

[Cour71] Courtois, P.J., Heymans, F., and Parnas, D.L. Concurrent control with "readers" and "writers". *Comm. ACM 14*, 10 (Oct. 1971), 667-668.

[Denn66] Dennis, J.B., and Van Horn, E.C. Programming semantics for multiprogrammed computations. *Comm. ACM 9*, 3 (March 1966), 143-155.

[Dijk68a] Dijkstra, E.W. The structure of the "THE" multiprogramming system. *Comm. ACM 11*, 5 (May 1968), 341-346.

[Dijk68b] Dijkstra, E.W. Cooperating sequential processes. In *Programming Languages*, F. Genuys (ed.), Academic Press, New York, 1968.

[Dijk75] Dijkstra, E.W. Guarded commands, nondeterminacy, and formal derivation of programs. *Comm. ACM 18*, 8 (Aug. 1975), 453-457.

[Dijk76] Dijkstra, E.W. *A Discipline of Programming*. Prentice-Hall, New Jersey, 1976.

[Dijk81a] Dijkstra, E.W. An assertional proof of a program by G.L. Peterson. EWD779, Feb. 1981, Nuenen, The Netherlands.

[Dijk81b] Dijkstra, E.W. Personal communication, October, 1981.

[Feld79] Feldman, J.A. High level programming for distributed computing. *Comm. ACM 22*, 6 (June 1979), 353-368.

[Flon76] Flon, L., and Habermann, A.N. Towards the construction of verifiable software systems. *Proc. ACM Conf. on Data, SIGPLAN Notices 8*, 2 (March 1976), 141-148.

[Floy67] Floyd, R.W. Assigning meanings to programs. *Proc Amer. Math. Society Symp. in Applied Mathematics 19* (1967), 19-31.

[Gele82] Gelernter, D., and Bernstein, A.J. Distributed communication via global buffer. *Proc. Symposium on Principles of Distributed Computing*, Ottawa, Canada, August 1982, 10-18.

[Good79] Good, D.I, Cohen, R.M., and Keeton-Williams, J. Principles of proving concurrent programs in Gypsy. *Proc. Sixth ACM Symp. on Principles of Programming Languages*, San Antonio, Texas, Jan. 1979, 42-52.

[Habe75] Habermann, A.N. Path expressions. Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, June 1975.

[Habe80] Habermann, A.N., and Nassi, I.R. Efficient implementation of Ada tasks. Technical Report CMU-CS-80-103, Carnegie-Mellon University, Jan. 1980.

[Hadd77] Haddon, B.K. Nested monitor calls. *Operating Systems Review 11*, 4 (Oct. 1977), 18-23.

[Hart77] Hartmann, A.C. *A Concurrent Pascal Compiler for Minicomputers. Lecture Notes in Computer Science 50*. Springer-Verlag, New York, 1977.

[Hoar69] Hoare, C.A.R. An axiomatic basis for computer programming. *Comm. ACM 12*, 10 (Oct. 1969), 576-580,583.

[Hoar72] Hoare, C.A.R. Towards a theory of parallel programming. In *Operating Systems Techniques*, Academic Press, New York, 1972.

[Hoar74] Hoare, C.A.R. Monitors: an operating system structuring concept. *Comm. ACM 17*, 10 (Oct. 1974), 549-557.

[Hoar78] Hoare, C.A.R. Communicating sequential processes. *Comm. ACM 21*, 8 (Aug. 1978), 666-677.

[Hoar81] Hoare, C.A.R. The emperor's old clothes. *Comm. ACM 24*, 2 (Feb. 1981), 75-83.

[Hold80] Holden, J., and Wand, I.C. An assessment of Modula. *Software—Practice and Experience 10* (1980), 593-621.

[Howa76a] Howard, J.H. Signaling in monitors. *Proc. 2nd International Conference on Software Engineering*, San Francisco, Oct. 1976, 47-52.

[Howa76b] Howard, J.H. Proving monitors. *Comm. ACM 19*, 5 (May 1976), 273-279.

[Jaza80] Jazayeri, M., et al. CSP/80: a language for communicating processes. *Proc. Fall IEEE COMPCON 80* (Sept. 1980), 736-740.

[Jone80] Jones, A.K. and Schwarz, P. Experience using multiprocessor systems—a status report. *ACM Computing Surveys 12*, 2 (June 1980), 121-165.

[Kaub76] Kaubisch, W.H. Perrot, R.H., and Hoare, C.A.R. Quasiparallel programming. *Software—Practice and Experience 6* (1976), 341-356.

[Keed79] Keedy, J.L. On structuring operating systems with monitors. *The Australian Computer Journal 10*, 1 (Feb. 1978), 23-27. Reprinted in *Operating Systems Review 13*, 1 (Jan. 1979), 5-9.

[Kell76] Keller, R.M. Formal verification of parallel programs. *Comm. ACM 19*, 7 (July 1976), 371-384.

[Kess77] Kessels, J.L.W. An alternative to event queues for synchronization in monitors. *Comm. ACM 20*, 7 (July 1977), 500-503.

[Kieb79] Kieburtz, R.B., and Silberschatz, A. Comments on "communicating sequential processes". *ACM Trans. on Prog. Languages and Systems 1*, 2 (Oct. 1979), 218-225.

[Kohl81] Kohler, W.H. A survey of techniques for synchronization and recovery in decentralized computer systems. *ACM Computing Surveys 13*, 2 (June 1981), 149-183.

[Lamp77] Lamport, L. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering SE-3*, 2 (March 1977), 125-143.

[LamL80a] Lamport, L. The 'Hoare logic' of concurrent programs. *Acta Informatica 14* (1980), 21-37.

[LamL80b] Lamport, L. The mutual exclusion problem. Op. 56, SRI International, Oct. 1980.

[Lamp82] Lamport, L., and Schneider, F.B. The 'Hoare logic' of CSP, and all that. Technical Report TR 82-490, Department of Computer Science, Cornell University, May 1982.

[Lamp79] Lampson, B.W., and Sturgis, H.E. Crash recovery in a distributed data storage system. Xerox Palo Alto Research Center, April 1979.

[LamB80] Lampson, B.W. and Redell, D.D. Experience with processes and monitors in Mesa. *Comm. ACM 23*, 2 (Feb. 1980), 105-117.

[Lamp81] Lampson, B.W. Atomic transactions. In *Distributed Systems—Architecture and Implementation*, Lecture Notes in Computer Science Vol. 105, Springer-Verlag, New York, 1981.

[Laue75] Lauer, P.E., and Campbell, R.H. Formal semantics of a class of high level primitives for coordinating concurrent processes. *Acta Informatica 5* (1975). 297-332.

[Laue78] Lauer, P.E., and Shields, M.W. Abstract specification of resource accessing disciplines: adequacy, starvation, priority and interrupts. *SIGPLAN Notices 13*, 12 (Dec. 1978), 41-59.

[Laue79] Lauer, H.C., and Needham, R.M. On the duality of operating system structures. *Proc. Second Int. Symp. on Operating Systems*, IRIA, Oct. 1978, reprinted in *Operating Systems Review 13*, 2 (April 1979), 3-19.

[Levi81] Levin, G.M., and Gries, D. A proof technique for communicating sequential processes. *Acta Informatica 15* (1981), 281-302.

[Lisk81] Liskov, B.L. On linguistic support for distributed programs. *Proc. IEEE Symposium on Reliability in Distributed Software and Database Systems*, Pittsburgh, July 1981. 53-60.

[Lisk82] Liskov, B., and Scheifler, R. Guardians and actions: linguistic support for robust, distributed programs. *Proc. Ninth ACM Symp. on Principles of Programming Languages*, Albuquerque, New Mexico, Jan. 1982. 7-19.

[List77] Lister, A. The problem of nested monitor calls. *Operating Systems Review 11*, 3 (July 1977), 5-7.

[Loeh77] Loehr, K-P. Beyond Concurrent Pascal. *Proc. of Sixth ACM Symp. on Operating Systems Principles*, West Lafayette IN, November 1977. 173-180.

[Lync81] Lynch, N.A. Multilevel atomicity—a new correctness criterion for distributed databases. Tech. Report GIT-ICS-81/05, Georgia Tech. May 1981.

[Mao80] Mao, T.W., and Yeh, R.T. Communication port: a language concept for concurrent programming. *IEEE Trans. on Software Engr. SE-6*, 2 (March 1980), 194-204.

[Misr81] Misra, J., and Chandy, K. Proofs of networks of processes. *IEEE Trans. on Software Engr. SE-7*, 4 (July 1981), 417-426.

[Mitc79] Mitchell, J.G., Maybury, W., and Sweet, R. Mesa language manual, version 5.0. Xerox Palo Alto Research Center Report CSL-79-3. April 1979.

[Nels81] Nelson, B.J. Remote procedure call. Ph.D. Thesis, Dept. of Computer Science Report CMU-CS-81-119, Carnegie-Mellon University, May 1981.

[Nyga78] Nygaard K., and Dahl, O.J. The development of the SIMULA languages. *Preprints ACM SIGPLAN History of Programming Languages Conference, SIGPLAN Notices 13*, 8 (August 1978), 245-272.

[Owic76a] Owicki, S.S., and Gries, D. An axiomatic proof technique for parallel programs. *Acta Informatica 6* (1976). 319-340.

[Owic76b] Owicki, S.S., and Gries, D. Verifying properties of parallel programs: an axiomatic approach. *Comm. ACM 19*, 5 (May 1976), 279-285.

[Parn72] Parnas, D.L. On the criteria to be used in decomposing systems into modules. *Comm. ACM 15*, 12 (Dec. 1972), 1053-1058.

[Parn78] Parnas, D. The non-problem of nested monitor calls. *Operating Systems Review 12*, 1 (Jan. 1978), 12-14.

[Pete81] Peterson, G.L. Myths about the mutual exclusion problem. *Information Processing Letters 12*, 3 (June 1981), 115-116.

[Reed79] Reed, D.P. Implementing atomic actions on decentralized data. Preprints for Seventh Symp. on Operating Systems Principles, Pacific Grove, Calif., Dec. 1979, 66-74.

[Refe80] *Reference Manual for the Ada Programming Language.* U.S. Dept. of Defense, July 1980.

[Reid80] Reid, L.G. Control and communication in programmed systems. Ph.D. Thesis, Dept. of Computer Science Report CMU-CS-80-142, Carnegie-Mellon University, Sept. 1980.

[Reif82] Reif, J.H., and Spirakis, P.G. Unbounded speed variability in distributed communications systems. *Proc. Ninth ACM Conf. on Principles of Programming Languages*, Albuquerque, Jan. 1982, 46-56.

[Ritc74] Ritchie, D.M., and Thompson, K. The UNIX timesharing system. *Comm. ACM 17*, 7 (July 1974), 365-375.

[Rope81] Roper, T.J., and Barter, C.J. A communicating sequential process language and implementation. *Software—Practice and Experience 11* (1981), 1215-1234.

[Schl81] Schlichting, R.D., and Schneider, F.B. An approach to designing fault-tolerant computing systems. Tech. Report TR 81-479, Dept. of Computer Science, Cornell University, Nov. 1981.

[Schl82a] Schlichting, R.D., and Schneider, F.B. Using message passing for distributed programming: proof rules and disciplines. Tech. Report TR 82-491, Dept. of Computer Science, Cornell University, May 1982.

[Schl82b] Schlichting, R.D., and Schneider, F.B. Understanding and using asynchronous message passing primitives. *Proc. Symposium on Principles of Distributed Computing*, Ottawa, Canada, August 1982, 141-147.

[Schn78] Schneider F.B., and Bernstein, A.J. Scheduling in Concurrent Pascal. *Operating Systems Review 12*, 2 (April 1978), 15-20.

[Schn82] Schneider, F.B. Synchronization in distributed programs. *ACM Trans. on Programming Languages and Systems 4*, 2 (April 1982), 125-148.

[Schw78] Schwartz, J.S. Distributed synchronization of communicating sequential processes. Tech. Report, Dept. of Artificial Intelligence, University of Edinburgh, July 1978.

[Shaw74] Shaw, A.C. *The Logical Design of Operating Systems.* Prentice-Hall, New Jersey, 1974.

[SIGO78] *Operating Systems Review 12*, 1 (Jan. 1978).

[Silb77] Silberschatz, A., Kieburtz, R.B., and Bernstein, A.J. Extending Concurrent Pascal to allow dynamic resource management. *IEEE Transactions on Software Engineering SE-3*, 3 (May 1977), 210-217.

[Silb79] Silberschatz, A. Communication and synchronization in distributed programs. *IEEE Trans. on Software Engr. SE-5*, 6 (Nov. 1979), 542-546.

[Solo79] Solomon, M.H., and Finkel, R.A. The Roscoe distributed operating system. *Proc. Seventh Symp. on Operating System Principles*. Pacific Grove, Calif., Dec. 1979, 108-114.

[Spec82] Spector, A.Z. Performing remote operations efficiently on a local computer network. *Comm. ACM 25*, 4 (April 1982), 246-260.

[Vand81] Van de Snepscheut, J.L.A. Synchronous communication between asynchronous components. *Information Processing Letters 13*, 3 (Dec. 1981), 127-130.

[Wels79] Welsch, J., and Bustard, D.W. Pascal-Plus—another language for modular multiprogramming. *Software—Practice and Experience 9* (1979). 947-957.

[Wels81] Welsch, J., and Lister, A. A comparative study of task communication in Ada. *Software—Practice and Experience 11* (1981), 257-290.

[Wirt77a] Wirth, N. Modula: a language for modular multiprogramming. *Software—Practice and Experience 7* (1977). 3-35.

[Wirt77b] Wirth, N. The use of Modula. *Software—Practice and Experience 7* (1977). 37-65.

[Wirt77c] Wirth, N. Design and implementation of Modula. *Software—Practice and Experience 7* (1977). 67-84.

[Wirt77d] Wirth, N. Towards a discipline of real-time programming. *Comm. ACM 20*, 8 (August 1977). 577-583.

[Wirt80] Wirth, N. Modula-2. Technical Report 36. Institut fuer Informatik, E.T.H.. Zurich, March 1980.

[Wulf71] Wulf, W.A., Russell, D.B., and Habermann, A.N. BLISS: A language for systems programming. *Comm. ACM 14*. 12 (Dec. 1971). 780-790.