Carnegie Mellon University
Software Engineering Institute

Language and System Support for Concurrent Programming

Michael B. Feldman
The George Washington University

April 1990

Preface

Capsule Description

This curriculum module is concerned with support for concurrent programming provided to the application programmer by operating systems and programming languages. This includes system calls and language constructs for process creation, termination, synchronization, and communication, as well as nondeterministic language constructs such as the selective wait and timed call. Several readily available languages are discussed and compared; concurrent programming using system services of the UNIX operating system is introduced for the sake of comparison and contrast.

Philosophy

In the last decade, the arena of concurrent programming has widened from "pure" operating system applications to a multitude of real-time and distributed programs.

Since the late 1960s, concurrent programming has been seen as necessary to the implementation of computer-related objects in operating systems, such as user processes and device drivers. The early constructs of concurrent programming came about in an effort to give the relatively small community of system-kernel programmers a set of operations to foster development of more reliable operating systems.

In the more recent, expanded view, concurrent programming is seen as an approach to achieving a more faithful representation of objects in the world being modeled by a program. Many programs are, in fact, simulations of some system in the physical world: an aircraft moving in space, a bank with tellers and customers. The physical world consists of many objects operating independently of one another, needing occasionally to synchronize, communicate, or refer to shared information. In this view, concurrent programming is useful because it provides a more natural mapping of these real-world objects to program segments. A more natural representation is, it is said, easier to code, debug, and maintain.

Since an increasing number of programs are best coded as concurrent ones, an increasing number of programmers need to learn to do

concurrent programming reliably and well. As concurrency has moved from operating systems outward to applications, the need to do concurrent programming has moved from a small circle of systems programmers to the much larger community of applications programmers.

Accordingly, attention is increasingly being paid to the problems of concurrency as seen by the application programmer. Modern operating systems such as UNIX, VMS, and MVS provide system-service libraries concerned with creation, scheduling, synchronization, and communication of concurrent processes. Further, recent programming languages such as Ada, occam, and Concurrent C provide language-level constructs for concurrency.

Scope

The purpose of this module is to provide a broad view of the world of concurrency as seen by the application programmer. While no attempt is made to cover all systems and languages, the various concurrency services of UNIX and five readily available languages are surveyed in some detail. The module provides the material needed to understand how concurrency is supported in modern languages and systems, with emphasis on the increased level of abstraction and portability provided by language-level constructs.

A revised version of the module is planned. This version will include more information on concurrent programming services provided by the VMS and MVS operating systems.

Author's Address

Comments on this module are solicited, and may be sent to the SEI Software Engineering Curriculum Project or to the author:

Michael B. Feldman
Department of Electrical Engineering and Computer Science
The George Washington University
Washington, DC 20052

Language and System Support for Concurrent Programming

Outline

1. Processes, Their States and Operations
- 1.1. Nondeterminism
- 1.2. Process States
- 1.3. Processes as a Type
  - 1.3.1. Process Creation and Activation
  - 1.3.2. Process Termination
  - 1.3.3. Process Scheduling
  - 1.3.4. Process Synchronization
  - 1.3.5. Interprocess Communication
  - 1.3.6. Nondeterministic Constructs

2. Supporting the Programmer's Interface to Concurrency
- 2.1. The Operating System Approach to Concurrency
- 2.2. The Language-Construct Approach to Concurrency
- 2.3. The Portable Library Approach to Concurrency

3. Concurrency at the Operating System Level
- 3.1. UNIX
  - 3.1.1. Process Creation and Activation
  - 3.1.2. Process Synchronization
  - 3.1.3. Interprocess Communication
  - 3.1.4. Process Scheduling
  - 3.1.5. Process Termination
  - 3.1.6. Nondeterministic Constructs

4. Concurrency Through Programming Language Constructs
- 4.1. Co-Pascal
  - 4.1.1. Process Creation and Activation

4.1.2. Process Termination
4.1.3. Process Scheduling
4.1.4. Process Synchronization
4.1.5. Interprocess Communication
4.1.6. Nondeterministic Constructs
4.2. Ada
4.2.1. Process Creation and Activation
4.2.2. Process Termination
4.2.3. Process Scheduling
4.2.4. Process Synchronization
4.2.5. Interprocess Communication
4.2.6. Nondeterministic Constructs
4.3. Concurrent C
4.3.1. Process Creation and Activation
4.3.2. Process Termination
4.3.3. Process Scheduling
4.3.4. Process Synchronization
4.3.5. Interprocess Communication
4.3.6. Nondeterministic Constructs
4.4. occam
4.4.1. Process Creation and Activation
4.4.2. Process Termination
4.4.3. Process Scheduling
4.4.4. Process Synchronization
4.4.5. Interprocess Communication
4.4.6. Nondeterministic Constructs

5. Concurrency Through Portable Libraries

5.1. Modula-2
5.1.1. Process Creation and Activation
5.1.2. Process Termination
5.1.3. Process Scheduling
5.1.4. Process Synchronization
5.1.5. Interprocess Communication
5.1.6. Nondeterministic Constructs

1. Processes, Their States and Operations

Annotated Outline

The basic "unit" of concurrent programming is the *process* (called *task* in some implementations). A process is an execution of a program or section of a program. Multiple processes can be executing the same (section of a) program simultaneously. A set of processes can execute on one or more processors; in the limiting case of a single processor, all processes are interleaved or time-shared on this processor.

1.1. Nondeterminism

A fundamental property of concurrent processes is *nondeterminism*, that is, at any time, it is not known what will happen next. This is true whether the implementation is by interleaving or true parallelism. In the former case, it is not known if an interrupt that causes process switching will occur next, causing a process other than the currently running process to run. In the latter case, the respective speeds of the various processors cannot be perfectly matched, so that it is often not known which processor will do its next instruction next. Put more formally, in a set of nondeterministic components, at least the activity evoked, but possibly even the final state, is not necessarily uniquely determined by the initial state [Dijkstra75].

Sometimes it is desirable that the final state of a computation be determinate even if the computation is not. For example, it is desirable that the sum of two sub-expressions be the same whether the left side or the right side is evaluated first. Causing the sub-expressions to be evaluated by separate processes introduces nondeterminism, but the result should be determinate.

A quotation from Edsger Dijkstra eloquently states the importance of nondeterminism:

"Having worked mainly with hardly self-checking hardware, with which nonreproducing behavior of user programs is a very strong indication of a machine malfunctioning, I had to overcome a considerable mental resistance before I found myself willing to consider nondeterministic programs seriously. ... Whether nondeterminacy is eventually removed mechanically—in order not to mislead the maintenance engineer—or (perhaps only partly) by the programmer himself because, at second thought, he does care—e.g. for reasons of efficiency—which alternative is chosen is something I leave entirely to the circumstances. In any case we can appreciate the nondeterministic program as a helpful stepping stone." [Dijkstra75]

1.2. Process States

A process can, during its lifetime, be in one of several *states.* These states are *elaborated*, *running*, *ready*, *blocked*, *completed*, and *terminated*. We adopt the set of states used in discussions of Ada tasks because this set is the most complete and general.

- A process is *elaborated* if its declaration (that is, the statements comprising the program or section of a program) has been processed and thus the process can be activated.
- A process is *running* if its statements are actually being executed by a processor.
- A process is *ready* if it is waiting only to be assigned to a processor.
- A process is *blocked* if it is waiting for some event to occur, such as an input/output completion, a timed wakeup, or a synchronization operation with another process. Sometimes the term *asleep* is used as

a synonym. Note that a process waiting only for assignment to a processor is not blocked, but ready.

- A process is *completed* if it has finished executing its sequence of statements but cannot yet terminate because it has active dependent processes. This state is of interest in environments such as Ada in which process termination always waits upon the termination of dependent processes.
- A process is *terminated* if it never was, or no longer is, active.

A process is *active* if it is running, ready, blocked, or completed; a process is *awake* if it is running or ready.

Figure 1 shows a state diagram for the process states.

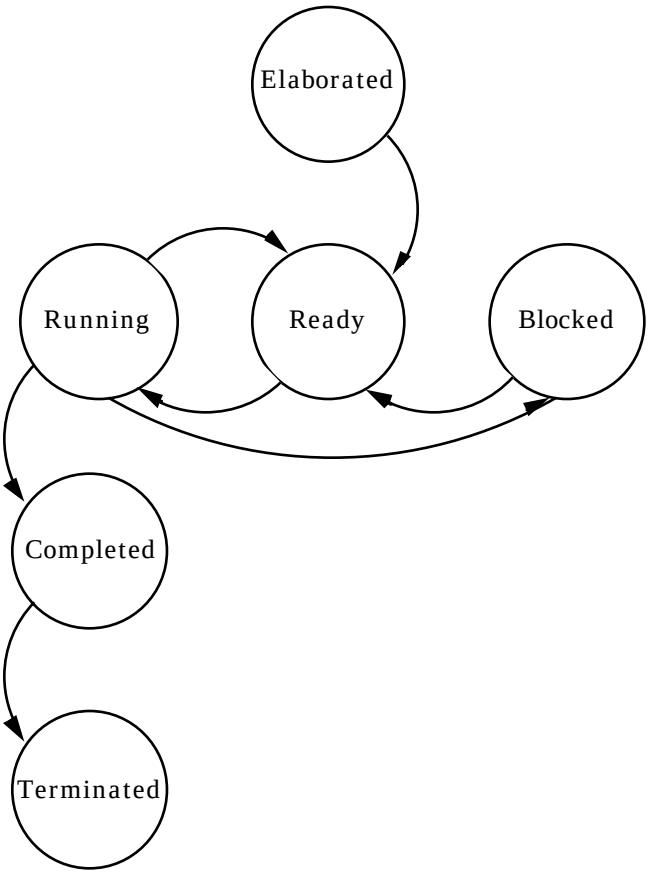

Figure 1. Process states

1.3. Processes as a Type

Current thinking about process management treats processes as a *type*. Recall that a type consists of a *set of values* and a *set of operations* (we consider the term *abstract data type* to be synonymous, and therefore will use the more concise term for brevity). Viewing processes in this way is consistent with the language view of processes, and gives us a canonical framework in which to examine both the system and the language views.

A number of operations are associated with processes. These are *creation and activation, termination, synchronization, interprocess communication,* and *scheduling*. In addition, programming languages often include *nondeterministic constructs* that affect the behavior of concurrent programs. We shall use this categorization, loosely based on that in

[Bal89], to examine the concurrency support at both the system and language levels.

1.3.1. Process Creation and Activation

We distinguish between creation and activation because in some systems and languages, processes can be declared (created) but not started (activated) until a later point in the creating program is reached. Further, in some cases the activation is implicit, occurring automatically when that point is reached, while in others the activation occurs only when an explicit operation is executed.

For our purposes, process *creation* is the elaboration of a declaration of a code segment that may eventually be activated. *Activation* is then the implicit or explicit operation that results in *an* execution of the process (*an* is emphasized because there may be many such executions).

The two main ways to activate processes are as follows:

1. Activating a process to execute a specific body of code while the activating process continues to execute.
2. Activating a list of processes to execute a list of statements concurrently while the activating process waits until the created processes are done.

The first allows activation of an arbitrary number of processes. The second activates only as many processes as there are statements, and if the activation statement cannot itself be activated more than once at the same time, say by recursion, the total number of processes that can be simultaneously active is bounded. The first way is, therefore, more general, but the second is easier to treat formally. Moreover, in practice, it is usually possible to structure a concurrent system so that only a fixed number of processes are needed.

The first way of activating a process is exemplified by the ability to fork a process in UNIX and the ability to declare and activate nested task objects in Ada. The second is exemplified by the **cobegin-coend** construct, which gives a list of statements to be executed concurrently.

1.3.2. Process Termination

How a process terminates differs from system to system and can be fairly complex. A process can be terminated implicitly, for example, if it "flows off the bottom" of its sequence of statements, or explicitly, by means of a *kill* (UNIX) or *abort* (Ada) operation.

In some languages (Ada, for example), process termination is structured in the sense that a process cannot terminate while it has active dependent or "child" processes; this makes it impossible for a process to disappear and leave its children without context. In such languages a process can express its *willingness* to terminate if and only if all the other conditions are right (for example, if its parent and siblings are also willing to terminate).

1.3.3. Process Scheduling

By process scheduling we mean the method(s) by which processes are assigned to processors. In other terms, scheduling is the means by which processes move from the ready state to the running state. In the event that there are fewer processors than processes, some sort of process scheduler must arbitrate among various processes contending

for scarce processor resources (in the limiting case, for a single processor).

Process scheduling is of course an important part of classical operating system design; with the advent of programmer interfaces to concurrency, and especially of languages like Ada designed to support concurrency in a system-independent way, process scheduling has become an important issue for language designers and implementors, and for applications programmers as well.

Considering the multiprocessor case, scheduling issues include whether the programmer can explicitly assign processors to processes, or whether this is the private domain of the process scheduler. In the case of contention for the processor(s), there are a number of additional issues to consider:

- Whether, and how, the programmer can assign scheduling priorities, and whether these priorities are statically determined or dynamically variable at run time.
- Given several contending processes of equal priority, how the manager determines which one to schedule, and whether the programmer can control this.
- Whether time-slicing is implemented, and if so, whether the programmer can control it in any way (here we use *time-slicing* to mean allowing each process a fixed quantum of time on the processor, after which it is interrupted and perhaps "swapped out").
- Whether a high-priority task moving from a blocked to a ready state (for example by the expiration of a delay or sleep operation, or completion of an I/O call), immediately interrupts the running process (preemptive scheduling), or whether the running process is allowed to run until it blocks (run-till-blocked scheduling).

1.3.4. Process Synchronization

There are two kinds of synchronization primitives:

1. *Unenforced* primitives, whose use to access shared objects, whether memory or a communication channel, is not enforced by the language or system.
2. *Enforced* primitives, whose use to access shared objects is indeed enforced by the language or system.

If use is enforced, then there is no way to access the object without that access being under the control of the synchronization primitives. This guarantees that access to the object is orderly. This fact makes proving assertions about programs much easier, as one does not have to worry about an object changing behind one's back, i.e., via an assignment by a process that chooses not to subject itself to access control.

Among the primitives of the unenforced kind are *semaphores* [Dijkstra68] and other devices that allow programming of critical regions. If these are properly used, one is guaranteed that at most one process can be in any critical region controlled by any one semaphore. If one has placed all uses of an object inside these critical regions, then one knows that at most one process can be accessing that object at any time. The problem with this approach is that one is not forced to put all accesses to the object inside these critical regions; there can be uses of such an object anywhere in the program.

Among the primitives of the second kind are *monitors* [Hoare74], which are basically abstract data type modules that guarantee that at

most one process at a time can be executing any procedure within any instantiation of the monitor. When combined with the compiler's protection that there be no use of a local variable of the monitor except within the monitor, one is assured that at most one process at a time can be accessing these variables.

Other unenforced primitives are UNIX's signals and similar structures in other operating systems. Other enforced primitives are Ada's entries, occam's communication channels, and UNIX's pipes.

Observe that monitors and Ada's entries can be misused to implement semaphores; in these languages, there is nothing that limits the programmer to use only variables local to the construct. Thus, there is nothing guaranteeing that a non-local variable of the construct is not also accessed from a place not inside the construct. However, if no such construct uses non-local variables (except possibly only to read them) there is no such problem, and proofs can be carried out without worrying about them. In a full-declaration, strongly typed language such as Ada, the compiler can detect all uses of non-local identifiers. Thus, there is the possibility of compile-time enforcement of limited access.

1.3.5. Interprocess Communication

With any method of creating processes there are two basic methods for processes to communicate:

1. *Shared variables:* the processes share memory and thus communicate by writing to and reading from any variable that both can access;
2. *Message passing:* the processes share no main memory and have to communicate by sending messages to each other via communication links accessed by each process as an input/output device.

The former can be used only when processes run on the same machine, while the latter can be used either when the processes run on the same machine or when the processes run on separate machines connected by a communication net. It is conceivable for shared variables to be used where processors do not share memory: in this case, the compiler or run-time system creates multiple physical copies of a single logical variable, and becomes responsible for making sure that all copies are brought up to date at the same time. The implementation difficulties in such a scheme are obvious.

The former method is more general in that the values communicated via variables can be information that is not really useful when transmitted from one machine to another, such as main memory addresses. However, this generality is accompanied by more opportunities for interference, race conditions, etc. These problems complicate any attempt to prove the behavior of the programs involved. Avoiding these problems in shared-memory concurrency requires a disciplined use of synchronization primitives, which are not easy to model formally. Moreover, there may be no guarantee that all access to these shared variables is via the synchronization primitives; thus, one must prove the absence of other accesses. The latter restricts communication enough so that all communication is via the message passing or input/output primitives. There is no chance for the problems of interference to occur. The greatly reduced chance for interaction allows modeling the processes as largely independent except for the explicitly announced interaction via message passing or input/-output.

Within the general scheme of message passing are two alternatives:

1. *Synchronous,* in which the sending process is blocked until the receiving process has accepted the message (implicitly or by some explicit operation).
2. *Asynchronous,* in which the sender does not wait for the message to be received but continues immediately. This is sometimes called a nonblocking or no-wait send.

Synchronous message passing, by definition, involves a synchronization as well as a communication operation. Since the sender process is blocked while awaiting receipt of the message, there can be at most one pending message from a given sender to a given receiver, with no ordering relation assumed between messages sent by different processes. The buffering problem is simple because the number of pending messages is bounded.

In asynchronous message passing, pending messages are buffered transparently, leading to potential unreliability in the case of a full buffer. For most applications, synchronous message passing is thought to be the easier method to understand and use, and the more reliable one as well. Asynchronous message passing allows a higher degree of concurrency.

Of the widely known and widely available languages supporting message passing, most (including CSP, occam, and Ada) implement a synchronous model. Concurrent C initially supported only synchronous message passing, but now permits both models.

A particular variety of synchronous message passing is the *rendezvous* of Ada and Concurrent C. Message types are called *entries* in Ada and *transactions* in Concurrent C. There is always a caller and a server, or sender and receiver; the sending operation is called a *call;* the receiving operation is called an *accept*. A process arriving at a call operation becomes blocked until the receiver is ready to accept the call. A server arriving at an accept operation becomes blocked if no call is pending. Each language allows both selective and timed call and accept operations to prevent indefinite blocking. When the caller and the server are both ready to communicate, a rendezvous takes place. During the rendezvous, data can be passed to and from the server, and the server may carry out a computation while the caller remains blocked. The rendezvous is properly called *extended* because an accept statement may involve extensive processing in order to construct the reply parameters [Burns85].

A particularly complete, lucid, and balanced comparative discussion of synchronous and asynchronous message passing can be found in Chapter 7 of [Gehani89]. To summarize the discussion there, synchronous message passing has the following advantages over asynchronous:

- Bidirectional information transfer, which is not possible in asynchronous message passing.
- Understandability, in that a sending process resuming execution knows that a message has been received.
- Clearer semantics, in that semantics do not depend on system resources (hidden buffering, for example).
- Easier error reporting, because sender waits for acknowledgement.
- Built-in synchronization.
- The possibility of timeout requests.

- Easier detection of problems such as deadlock; in asynchronous implementations, these conditions may arise only when the system is “stressed”.
- Ease of implementation.

The advantages of asynchronous message passing are:

- Maximum flexibility and expressiveness for the programmer.
- The possibility of pipelining message transmission.
- Easy simulation of synchronous message passing.
- More efficient implementation, if information flow in the application is unidirectional.

1.3.6. Nondeterministic Constructs

The way processes interact is often not deterministic, but rather is determined as execution progresses. A process may wish to select other processes to interact with, depending on its own state and on pending requests for interaction from other processes. For example, a buffer process may accept a request from a producer to store an item whenever the buffer is not full. It may accept a request from a consumer to supply an item whenever the buffer is not empty. And if both requests arrive simultaneously, it must decide between them. To program behavior like this, a notation is needed to express and control nondeterminism [Bal89].

Some communication operations are nondeterministic: a message received indirectly through a port may have been sent by any process. Such an operation expresses nondeterminism, but does not control it.

Many languages provide a construct for controlling nondeterminism; this construct, based on Dijkstra’s guarded command [Dijkstra75] is usually called a *select statement*. A select statement usually consists, in some syntactic form, of a list of guarded commands of the form

guard –> statements

A guard consists of a boolean expression followed by some sort of communication request. Upon entry to the select, the guards are evaluated and the commands are partitioned according to whether the guards are true or false. Those with false guards are ignored; among those with true guards, one is selected nondeterministically.

The preceding paragraph is an oversimplification; each language supporting select statements has its own details and idiosyncracies. Languages with select statements include occam, Ada, and Concurrent C.

2. Supporting the Programmer’s Interface to Concurrency

Since operating systems began in the 1960s to support concurrent processes, some means have been provided to the programmer (if only the system programmer) to create and manage them. Over time, these means have fallen into three categories:

1. Libraries of supervisor calls to proprietary operating systems.
2. Special (syntactically distinct) language constructs, independent of the operating system or target computer.
3. Language constructs at the semantic level only; that is, subroutine libraries supporting process types and operations, independent of the operating system or target computer.

Examples of the first approach are the system-call libraries of VM and MVS (IBM mainframe operating systems), and VMS (DEC VAX minicomputer operating system). Example languages embodying the second approach are Concurrent Pascal, Ada, Concurrent C, and occam. The best example of the third approach is the process module of Modula-2.

The interrelationship between the C programming language and the UNIX operating system is something of a special case. On the one hand, C's support for concurrency is certainly at the level of supervisor calls; on the other, UNIX itself is such a portable (and widely-ported) operating system that, in a very real sense, concurrent programming in UNIX can be independent of a hardware platform. Indeed, a portable specification of process management in UNIX is specified in the Portable Operating System Implementation Standard (POSIX) [IEEE88].

However, C is available and widely used on non-UNIX operating systems, for example MS-DOS, where the concurrency library is quite different. Thus we consider the C/UNIX connection to be more in the first category than the third.

2.1. The Operating System Approach to Concurrency

Not much can be said in a general way about concurrency support via system calls. Typically these are available either from assembly language or from a high-level language; by definition, programs using them are entirely non-portable because they are tied explicitly to the operating system.

Proponents of this approach argue that it is the only way to achieve acceptable run-time efficiency, especially in real-time systems, because a system-independent approach to processes does not map cleanly enough onto the underlying system-level processes. Opponents argue that this approach perpetuates hardware-first design and causes expensive software systems to be rewritten almost from scratch just to adapt them to emerging hardware or system technologies.

It is also the case that because system-level concurrency primitives are at a lower level of abstraction than language-level ones, concurrent programs using system services are more complex, less obvious, and (probably) more difficult to debug.

2.2. The Language-Construct Approach to Concurrency

As concurrent programming is increasingly seen as having broader application than the construction of system kernels, attention is increasingly being given to designing programming languages so that they support concurrent programming. Emphasis is placed on building structures into the languages, instead of just providing supervisor calls in the form of subroutine libraries. There are three important reasons for creating special language structures for concurrency:

1. *Maintainability.* Concurrency is expressed in a way that is clearly visible to the writer and reader, rather than being hidden in the bodies of the called subprograms. Often the concurrent operations follow the same sort of block structure as the sequential operations; their use results in a program which is better structured and therefore easier to test and maintain.
2. *Abstraction.* If concurrency is being used to model objects in the physical world, then this modeling is clearer and more natural if the concurrent program segments follow an identifiable structure that distinguishes them from sequential segments.

3. *Portability and Machine-Independence.* A program will be less machine-dependent if all structures in it, including those implementing concurrency, are designed independently of a given operating system or hardware platform.

As seen above, arguments against the use of language-level constructs typically emphasize the performance penalties often associated with machine-independence. Indeed there may be some short-term tradeoffs, but these are no different from the short-term penalties we have paid historically whenever functionality has been moved from a lower level to a higher one. In the longer term the penalties diminish: with language maturity comes greatly increased optimization by the compiler, and often the hardware evolves to support software needs. Particularly useful historical examples of this evolution are:

- The greatly increased efficiency of procedure calls generated by modern compilers and also the nearly universal hardware support for run-time stacks.
- Universal availability of hardware support for arithmetic: in older computers, not only floating-point computations, but, in some cases, even multiplication and division were done purely in software. Today's users of personal computers can trade cost for performance in considering whether to purchase math co-processor chips.

There is no reason to doubt that concurrent programming will fit into this historical scheme. Evidence for this is seen in the Ada context, where compilers already have evolved so that task operations have been greatly speeded up, and in the trend toward hardware support for context switching.

An argument more difficult to counter is the one that a rigid, fixed set of syntactic constructs is closed and cannot be modified or extended without imposing a major change on the language definition and all existing compilers. The only counter-argument is that this inflexibility is no worse than that found in any other set of high-level language features; by definition, high-level features must be described in language reference manuals and recognized by compilers. A change to *any* feature of a high-level language is costly.

2.3. The Portable Library Approach to Concurrency

A compromise position between the two extremes of operating systems primitives and programming language structures can be found in the portable-library approach to concurrency. With this approach, no special syntactic structures are imposed on the language. Instead, a library or module supports concurrency by defining types (*process*, for example), and operations in the form of subprograms that can be called from an application. The programmer interface to such modules is machine- and system-independent; the implementation of the modules, of course, depends upon the underlying operating system and hardware.

Proponents of this approach argue that it embodies all the virtues of machine-independence and portability, and yet is open in the sense that a module can be extended without imposing any compiler reimplementation costs. This is certainly true, but if the syntactic nature of concurrency is missing, so is the high-level abstraction and visible use of concurrency. Recent efforts to standardize a set of concurrency primitives for Modula-2 through two library modules will help alleviate this problem [BSI89].

3. Concurrency at the Operating System Level

3.1. UNIX

UNIX [IEEE88, Kernighan84, Rochkind85] is not a single operating system, but an entire family of operating systems. Specifically, the constructs for concurrent programming differ somewhat between the System V subfamily and the Berkeley subfamily; and at the detail level there are even vendor-to-vendor or version-to-version differences within subfamilies. The discussion here is therefore necessarily oversimplified to avoid focusing on the differences and to provide a "portable" view of UNIX. The discussion is based chiefly on the POSIX standard [IEEE88], which describes the nearest thing to a common, portable, UNIX programmer's interface. Even this is not entirely portable, because none of the current UNIX systems implements precisely the POSIX standard.

3.1.1. Process Creation and Activation

UNIX uses a pair of system calls, *fork* and *exec*. The fork call creates a copy of the forking process, but with its own address space. The exec call is invoked by either the original or copied process to replace its own virtual memory space with a new program, which is load into memory, destroying the memory image of the calling program.

3.1.2. Process Synchronization

Some versions of UNIX implement semaphores. The only common synchronization mechanism is the *signal*. Signals fall into several functional categories, ranging from interrupt-like signals to a process-abortion command.

3.1.3. Interprocess Communication

Shared memory is implemented in some versions of UNIX. The only form of communication common across versions is the *pipe*. A pipe may be created before a fork; its process endpoints are then set up between the fork call and the exec call. A pipe is essentially a queue of bytes between two processes; one writes into the pipe, and the other reads from the pipe. The size of the pipe is fixed by the system (typically 4096 bytes). Reading from an empty pipe or writing to a full one causes the process to be blocked until the state of the pipe changes.

3.1.4. Process Scheduling

No single process scheduling scheme appears across UNIX versions; the POSIX standard does not specify one. It can be assumed that in particular versions, process priorities can be assigned and time-slicing is implemented.

3.1.5. Process Termination

A process terminates by using the *exit* system call; its parent process can wait on the termination event by using the *wait* system call.

3.1.6. Nondeterministic Constructs

UNIX does not provide an explicit construct for expressing or controlling nondeterminism.

4. Concurrency Through Programming Language Constructs

In surveying the concurrent programming facilities of a number of programming languages, this module has focused on languages likely to be available to students and teachers for experimentation. The instructor or student interested in a comprehensive survey of languages for concurrent programming, including quite thorough coverage of current research, is referred to the excellent paper by Bal, Steiner, and Tanenbaum [Bal89].

We have chosen four languages for comparison in this section: Co-Pascal, Ada, Concurrent C, and occam. A discussion of Modula-2 appears in the next section, as concurrency in Modula-2 is really based on portable libraries rather than language primitives.

As in the operating systems section, we have followed an organizational model in which, for each language discussed, a background and historical section is followed by descriptions of the concurrent programming facilities in the language, grouped according to the major categories of operations introduced in Section 1.3. The discussion is not intended to teach the languages, but rather to orient the reader to the facilities and important literature. Accordingly there is almost no discussion of syntax or presentation of code segments; rather, there is a prose description of the various operations.

4.1. Co-Pascal

Co-Pascal is the concurrent language of the examples in the textbook by Ben-Ari [Ben-Ari82]. Included with the book is a Pascal source listing of a portable implementation of this language, built as a modification of the Pascal-S compiler/interpreter of Niklaus Wirth. Versions exist for several timesharing computers, and a version was produced by Schoening [Schoening86] for the IBM Personal Computer family.

Co-Pascal is a subset of Pascal (sets, pointers, and dynamic allocation are missing) with the addition of concurrent programming constructs as described below. The recursive-descent compiler produces P-code instructions that are interpreted.

This is an interesting system with which to begin the study of concurrency, because even though it is essentially an educational system of limited capacity and it does not compile genuine machine code, it embodies the **cobegin-coend** style of creating and activating processes, as well as a multivalued semaphore type with *wait* and *signal* operations. Most interesting for students, the process scheduler employs time-slicing (where the quantum is measured in P-code instructions) and scheduling by random selection among the ready processes. This nondeterminism serves as an excellent way to convince students of the need for mutual exclusion and the benefits of concurrency as an abstraction mechanism.

Even on a single-processor computer (including a personal computer), the problems of concurrency can be clearly illustrated and solutions tested with very little overhead but with a realistic execution model.

4.1.1. Process Creation and Activation

Procedures with parameters can be activated simultaneously as processes by enclosing their invocations between a **cobegin-coend**

pair. The same procedure can be associated with multiple processes; this is useful in illustrating the difference between a procedure called sequentially and one activated as one or more processes. It is not possible to nest **cobegin-coend** pairs. Formally, when a **cobegin** is encountered, all procedure invocations are created as processes; they are all activated when the matching **coend** is reached; the main program is suspended while processes are active.

4.1.2. Process Termination

A process terminates when control reaches the end of its statements. Execution of the main program resumes when all processes within a **cobegin-coend** pair have terminated.

4.1.3. Process Scheduling

The scheduler allows the running process to execute a randomly chosen number of P-code problems or to block on a semaphore, whichever comes first. At this point, a random selection is made among the ready processes, and the selected process is resumed.

4.1.4. Process Synchronization

Synchronization is done by means of semaphores with wait and signal operations. A semaphore is just an integer variable; wait(s) and signal(s) are compiled as procedures passing the address of the parameter. The wait operation is implemented as:

```
if s > 0
   then s := s - 1
   else SUSPEND := address of s
            {SUSPEND is a field in the control block of each process}
```

The signal operation is implemented as:

```
search for an active process with SUSPEND = address of s
if one is found
   then set its SUSPEND field to 0
   else s := s + 1
```

Ben-Ari notes that this is a starvation-free implementation suggested by Morris [Morris79].

4.1.5. Interprocess Communication

Interprocess communication is done exclusively by means of shared variables; no message-passing or other abstraction mechanism is provided.

4.1.6. Nondeterministic Constructs

No nondeterministic constructs are provided, but unpredictable execution is assured by the randomness of the process selection algorithm in the scheduler.

4.2. Ada

The language that became Ada was chosen in a competition sponsored in the late 1970s by the U. S. Department of Defense (DoD); Ada is intended to become the language in which new defense-related software is written. Concurrent programming primitives were included in the language because the various DoD requirements documents (“Steelman” and its predecessor reports) listed concurrent programming as essential to the mission of the language. Standardized in 1983, before any viable compil-

ers existed, Ada was the first language intended for widespread industry adoption and use to include a full set of concurrent programming primitives [DoD83, Nyberg89, Ichbiah86, Gehani84, Burns85, Shumate88].

At this writing, approximately 300 different compilers have been validated (tested for conformance to the standard) by the government; compilers exist for virtually all current computers. The suite of validation test programs, called the Ada Compiler Validation Capability, continues to evolve slowly; a new version goes into effect every eighteen months.

A process in Ada is called a *task*. Each task is actually an object of a *task type;* a task type is declared by the programmer; the programmer can, in the limiting case, simply declare individual tasks, a type for each of which is implicitly declared by the compiler. Such tasks are said to have *anonymous* types. A task type declaration consists of a *specification* and a *body*. The specification consists of declarations of the entries (messages) that can be called and accepted; each entry has an associated parameter list, similar to that for a procedure. The task body has the structure of an ordinary block, complete with its own declarations. Since task bodies are active code, a typical task body is written as an infinite loop.

4.2.1. Process Creation and Activation

A task in Ada is created in one of two ways: by declaration or by dynamic allocation. Simply declaring one or more variables of a given task type creates the tasks; it follows that arrays of tasks of a given type can also be declared and created. It is also possible to declare a pointer type capable of pointing to a task object; a call to the allocator operation **new** dynamically creates an object of the task type, returning a pointer to it in some pointer variable. In supporting task types and declared or dynamically allocated task objects, Ada treats processes in a manner analogous to data structures.

Ada is a block-structured language, so a task object is declared in the declaration part of some **begin-end** block. A task object so declared is activated (and begins execution) just before the statements of the block are executed. If more than one task is declared in a given block, the language standard [DoD83] states that they are activated "in an order not defined by the language." The declaring program unit continues execution as an independent task. Note that no explicit task activation operation is required or provided; activation is implicit.

A task object dynamically created by an allocator call is activated just after its creation; execution of the creating program unit then proceeds as an independent task. This is roughly analogous to a fork operation. It is a consequence of this dynamic creation capability that the total number of active processes is bounded only by available memory, not by anything in the language.

4.2.2. Process Termination

A task always has a *master* on which it depends. Intuitively, the master is the program block in which the task is declared. A task can also be declared in a library package, in which case the package is the master. A dynamically activated task depends upon the block in which the pointer type is declared, not upon the block in which the allocating statement appears.

Task termination in Ada is designed to follow this dependency. A task that reaches the end of its sequence of statements is said to be

completed. A task terminates [Gehani84, p. 46] if any of the following conditions hold:

- It has completed and it has no dependent tasks.
- It has completed and all dependent tasks have terminated.
- It is waiting at a terminate alternative (see discussion of the select statement below), its master is completed, and all its siblings (other tasks dependent on the same master) have either terminated or are waiting at a terminate alternative.

Termination is intended to be reliable in that a task cannot terminate and leave a dependent child with no context or master.

The Ada standard does not specify whether tasks declared in library packages must ever terminate; since a library package is global to a main program importing it, conceptually a main program can terminate while library tasks continue to execute. This is a potentially useful concept in the case of background tasks whose execution persists across several program executions.

4.2.3. Process Scheduling

The Ada tasking model is supposed to be rich enough to be implemented on any number of underlying machine architectures, from simple embedded processors to personal computers to huge computer networks. Accordingly, the precise algorithm for distributing tasks over processors, or indeed for scheduling tasks on a single processor, is not defined by the language.

Ada supports a weak static scheduling priority scheme: a task may be given a priority at compilation time, by means of a compiler directive. All objects of the same task type have the same priority. The number of priorities is not defined by the language; the limiting case of a single priority (i.e., no differentiated priorities at all) is allowed. Section 9.8 of the Ada language reference [DoD83] says the following about priority:

> The specification of a priority is an indication given to assist the implementation in the allocation of processing resources to parallel tasks when there are more tasks eligible for execution than can be supported simultaneously by the available processing resources. ...
>
> If two tasks with different priorities are both eligible for execution and could sensibly be executed using the same physical processors and the same other processing resources, then it cannot be the case that the task with the lower priority is executing while the task with the higher priority is not.
>
> For tasks of the same priority [or tasks without explicit priority] the scheduling order is not defined by the language. ... If the priorities of both tasks engaged in a rendezvous are defined, the rendezvous is executed with the higher of the two priorities. ...
>
> Priorities should be used only to indicate relative degrees of urgency; they should not be used for task synchronization.

The Rationale [Ichbiah86] states that the priority rules require preemptive scheduling if the implementation supports more than one priority. No other scheduling rules are specified: time-slicing is not required; indeed, if all active tasks have the same priority, run-till-blocked scheduling is perfectly legal. Note that as of this writing, not all

implementations that support priorities do preemptive scheduling; the evolving validation suite apparently does not test this yet.

A great deal has been written about Ada's task scheduling (or lack of it). Some writers in the real-time systems industry have been especially bitter critics of the lack of programmer control. A well-written critique of task scheduling appears in [Burns85], and [Cornhill87] is a typical example of real-time concerns.

4.2.4. Process Synchronization

Synchronization in Ada is done by means of the rendezvous. Acalling task arriving at an unconditional entry call in some server task is forced to block until the call is accepted. The language specifies that pending calls on a given entry are placed on a strict first-in, first-out queue. A server task arriving at an unconditional accept statement blocks if there are no pending callers. Calls and accepts need not be unconditional, however; they can be selective or timed on both the caller and server sides. See the discussion below of nondeterministic constructs.

4.2.5. Interprocess Communication

Ada supports both shared-variable and synchronous message communication. A task, like a sequential subprogram in Ada, is allowed to make non-local references to any variables which are visible to it. Thus two tasks to which the same non-local variable is visible may both access it. Such access is not protected by the language.

This is obviously not the recommended way to do task communication; it is better to communicate values via entry parameters during a rendezvous. Entry parameters may be passed to or from the server, or may be bidirectional. The recommended way to access a shared data structure is to encapsulate that structure in a server task which acts as a monitor, using entries to support read and write access rights.

4.2.6. Nondeterministic Constructs

Ada uses a form of guarded commands called the *select* statement. A select statement consists of a list of (possibly guarded) *accept* alternatives. A *guard* is a boolean expression. Upon arrival at a select, all guard expressions are evaluated; then the alternatives are divided into an open set and a closed set, according to whether their guards are, respectively, true or false. Among the open alternatives, a single call is accepted. If calls are queued on several entries for which corresponding accept alternatives are open, "one of them is accepted arbitrarily (that is, the language does not define which one)." [DoD83, sect. 9.7.1].

The use of the word *arbitrarily* has been the cause of much controversy. In fact, the language of the standard does not force an arbitrary (i.e., nondeterministic or random) selection; what is arbitrary is that the standard does not require a particular algorithm for the selection, i.e., the manner of selection is implementation-dependent. The selection need not be fair; indeed, an official commentary [Nyberg89, sect. 9.7.1] points out that the priorities of the callers at the heads of the various queues may, but need not be, taken into account. The Ada Rationale [Ichbiah86] recommends that the selection method not be particularly predictable, advocating that using guards to force predictability is better design strategy because it is more explicit.

When an alternative is selected, the accept and possible subsequent statements are executed. During execution of the accept, i.e. during the rendezvous, the caller remains blocked. If no callers are queued on any of the open alternatives, the behavior of the select statement depends upon whether one of several other (mutually exclusive) alternatives is present:

- An *else* clause, only one of which may be present on a select. If it is present, it is selected if no callers are queued on open accepts, or if no alternatives are open. The else clause may consist of a sequence of statements to be executed.
- A (possibly guarded) *delay* alternative, which serves as a timeout. The operand of a delay is in units of elapsed seconds; if an open delay alternative is present, the server waits at the select until either the delay expires or a caller arrives at an open accept, whichever comes first. There may be more than one delay alternative; this is sensible if they are guarded to control which of several delay periods is to apply.
- A (possibly guarded) *terminate* alternative, of which only one may be present. This alternative may (only) be selected under the conditions described above under process termination.

So much for the server side. The caller is also not restricted to an unconditional entry call. There is a *selective call* statement, which enables a caller to withdraw its call and leave the queue if the server is not immediately able to accept the call, and a *timed call*, in which the call is withdrawn if it cannot be accepted within a given delay. Beginners are often confused by the fact that the caller-side selective and timed calls are also formed as "stripped-down" select statements, but the semantics are clearly different from the server-side select.

4.3. Concurrent C

Concurrent C [Gehani89] was developed in the middle 1980s by Narain Gehani and William Roome of AT&T Bell Laboratories. Available for several years only to researchers, and currently implemented on VAX, Sun, and AT&T computers, Concurrent C has become a commercial product for which, at this writing, implementations are expected for DOS and OS/2 target environments.

Concurrent C is an upward-compatible extension of C, with the addition of a concurrency model that is quite similar to Ada's, but with a number of improvements. It has been implemented as a C preprocessor plus run-time support for process management.

Gehani states, "We picked C as the basis for our work on parallel programming because (a) it is an immensely popular language, (b) it does not have parallel facilities, and (c) we use it. We had several objectives in enhancing C with concurrent programming facilities:

1. To provide a concurrent programming language that can be used for writing systems on genuinely parallel hardware, such as a network of microprocessors or workstations.
2. To provide a test bed for experimenting with a variety of high-level concurrent programming facilities.
3. To design a practical concurrent programming language that can be implemented on a variety of currently available multiprocessor architectures.

"C++ is an extension of C that provides data abstraction facilities [Stroustrup86]. Concurrent C does not provide data abstraction facilities

but its parallel programming facilities can be used in conjunction with C++. The Concurrent C compiler, as a compile-time option, accepts C++." [Gehani89]

Most of the terminology of Concurrent C conforms to that of Ada; the main syntactic differences are that a task in Ada corresponds to a *process* in Concurrent C, and an entry in Ada corresponds to a *transaction* in Concurrent C. Since the concurrency model is based on Ada but improves upon it, the discussion that follows focuses on the differences from Ada.

4.3.1. Process Creation and Activation

All processes in Concurrent C are objects of a process type; there is nothing corresponding to Ada's directly declared tasks of anonymous type. Whereas most tasks in Ada are implicitly activated, all processes in Concurrent C are explicitly activated at the time they are created by a create operation; the value returned by the create can be stored in a process variable, but need not be. In Ada, then, tasks of anonymous type are possible but tasks almost always have identifiers (or array elements) naming them; in Concurrent C there are no anonymous process *types*, but there can be anonymous process *objects* if the value returned by the create operation is discarded. In this case, of course, the process cannot be referred to or its transactions called.

Another important difference in Concurrent C is that as a process object is created, parameter values can be passed to it. An Ada task cannot have parameters supplied at creation or activation time, so these values must be explicitly passed during a rendezvous.

4.3.2. Process Termination

Process definitions in Concurrent C cannot be nested; this conforms to C's philosophy that function definitions cannot be nested. This makes termination somewhat simpler than in Ada, though the basic idea is the same. The difference is that in Ada a subset of the tasks can be made to terminate collectively (using the terminate alternative) while others continue to run. In Concurrent C the terminate alternative can be taken *only* if all the processes in the program have completed or are waiting at a select statement with a terminate alternative. Because of the block structuring in Ada, termination is more complex and the semantics are harder to define. In addition, the mechanism is harder to implement in a distributed environment because it is hard to get a consistent snapshot of the state of all processes on all processors. In Concurrent C, execution of the terminate alternative leads to termination of the entire program.

4.3.3. Process Scheduling

Concurrent C permits different process objects of the same type to have different priorities; further, dynamic alteration of scheduling priorities is permitted using a set of function calls to modify priorities.

The Concurrent C *create* operation has an optional clause allowing the programmer explicitly to specify a processor upon which to activate the created process. There is nothing defined in the Ada standard to accomplish explicit process/processor binding, although multiprocessor Ada implementations can provide for this using implementation-dependent compiler directives.

4.3.4. Process Synchronization

The Concurrent C rendezvous model is very similar to that of Ada, with an important difference: in Concurrent C the entry queues need not be managed in strict first in, first out manner. Specifically, an *accept* statement in Concurrent C may have a *suchthat (condition)* clause, which results in the queue being traversed until a pending call is found for which the specified condition is true. Also, an accept statement can have a *by (expression)* clause, in which case the expression is evaluation for each pending call and the call with the minimum value is accepted. The *suchthat* and *by* clauses obviously result in a priority-queueing mechanism for accept statements. This feature responds to what many argue is a serious liability in the Ada model.

4.3.5. Interprocess Communication

Normally, transactions are synchronous as in Ada. As a (recently added) option, a transaction can be declared as asynchronous, which allows a caller to continue its execution immediately after making the call. The language designers admonish the programmer to use asynchronous transactions only when synchronous ones are not appropriate, and they point out that asynchronous transactions are not necessarily faster than synchronous ones but often require much higher overhead.

An asynchronous transaction is appropriate when the server task is doing a lot of other work and accepts transaction calls only rarely. In this case, a caller might be blocked for a long time waiting for service. Another use might be where transmission delays are long, for example on a loosely coupled network.

4.3.6. Nondeterministic Constructs

Concurrent C's *select* statement is almost identical to Ada's. The only change is that there is no explicit *else* clause. Instead, Concurrent C provides for a (possibly guarded) block of statements to be executed if no other alternative can be immediately selected. Ada's *else* clause cannot be guarded, so this facility is an useful addition in Concurrent C.

4.4. occam

The programming language occam (the name is indeed written in lower case) was developed in the early 1980s in the United Kingdom [Pountain87]. It was intended to be a "high-level assembly language" for the Transputer chip, the main processor component for a highly parallel computer system. The language can be thought of as an implementation of CSP, though the notation is changed to a scheme more palatable to many programmers than the highly mathematical notation of CSP [Hoare78]. occam is interesting to study because it is gaining in popularity as a language for programming parallel computers. An occam system, including a Transputer simulator, is widely available under UNIX and is easily ported to other UNIX target computers.

4.4.1. Process Creation and Activation

Formally, occam treats every statement, including assignment statements and the like, as a process. A sequence of sequential statements is treated formally as a sequence of process creations and terminations. Creation and activation of processes are therefore

trivial operations which appear to be the execution of what are usually thought of as statements.

Processes in occam are built up from sequential and parallel compositions of other processes, starting with the three primitive processes of assignment, input from a channel, and output to a channel. Composition is done by the SEQ and PAR operations.

Source text in occam is not "free-format" as in most other high-level languages: block structure is imposed by indentation. The reserved word SEQ, followed by an indented series of statements, can be thought of as similar to the **begin-end** pair identifying a block in classical Algol-like languages: the statements following the SEQ are executed in sequence. Similarly, the reserved word PAR followed by an indented series of statements corresponds to the **cobegin-coend** pair of Concurrent Pascal: these statements (or *processes* in occam's terminology) are executed in parallel. Deeper nesting is imposed by deeper indentation; nesting can be to arbitrary depth, including nested PARs.

Arrays of processes can be created by a *replicator* operation, syntactically similar to a counting loop statement in other languages.

4.4.2. Process Termination

A process in occam terminates (only) when it has finished its work, that is, when controls flows "off the bottom." There is no special operation corresponding to the terminate alternative in Ada or Concurrent C, nor any special way to send a terminate message to a process. The occam primitive process STOP is not a terminate instruction, rather just the opposite: STOP blocks forever (a friendly implementation might detect this situation and write a diagnostic).

4.4.3. Process Scheduling

Parallel processes can, but need not, be explicitly allocated to processors by the PLACED PAR operation. In the absence of the PLACED qualifier, the implementation determines the allocation (in the limit, as always, a single processor).

Assuming that there are fewer processors than processes, scheduling priorities can be assigned to parallel processes in a very simple fashion: if the processes are created by a PRI PAR operation, scheduling priority is directly related to the order in which the processes are written down: a process appearing earlier in the list has a higher priority than one appearing later. If P and Q are concurrent processes with priorities p and q such that $p < q$, then Q is only allowed to proceed when P cannot proceed (e.g., is blocked on a delay or an input or output operation). The authors of the occam text advise very strongly that programs should be designed without recourse to a PRI PAR; this advice is reminiscent of that encountered among Ada authors who argue that portable concurrent programs should be written so that they are independent of optimization details like scheduling priorities.

In the absence of delays or I/O operations, scheduling appears to be implementation-dependent, as in Ada. In the University of Loughborough compiler for UNIX systems—which is readily available for teaching purposes—the occam program, with all its processes, is created as a single UNIX process, and the occam internal scheduler is invoked whenever a process loops or branches.

4.4.4. Process Synchronization

Synchronization in occam is done by means of operations that perform input from or output to channels. Two processes synchronize only to communicate over a channel.

4.4.5. Interprocess Communication

The only interprocess communication in occam is unidirectional synchronous message passing over channels. A channel is set up between two processes; the sending process can only output to the channel; the receiving process can only input from that channel. A channel can have a type called a *protocol*, which is a composite type analogous to a record. The protocol defines the form of a message. No buffering is provided by the language; it is up to the programmer to code processes which act as buffers.

Communication by shared variables is not allowed in occam: if one component of a PAR assigns to or inputs to a variable, this variable cannot be used by any other component of the PAR. The basic principle is that variables are used for storing values while channels are used for communicating values.

4.4.6. Nondeterministic Constructs

The guarded command is used in occam as a nondeterministic construct. The basic structure is an ALT keyword followed by a set of alternatives, each of which consists of an input statement optionally preceded by a boolean guard, and a series of statements (processes) to execute if the alternative is selected. An alternative can proceed if its guard is true and input is available from the channel. The occam language definition says “an alternation behaves like any one of the alternatives which can proceed, and can proceed if any of the alternatives can proceed.” As in Ada, if two or more alternatives can proceed, the selection mechanism is not defined by the language and is thus implementation-dependent.

5. Concurrency Through Portable Libraries

5.1. Modula-2

Modula-2 as described in its reference manual [Wirth85] provides only a few coroutine primitives. Modula-2's author, Niklaus Wirth, belongs to what might be called the “small is beautiful” language school, arguing against high-level concurrency operations. In a 1984 paper [Wirth84] he asserts that since “genuine concurrency” does not exist in a single-processor environment, coroutines, the basic mechanism for switching between logical processes, are sufficient.

Wirth conceived Modula-2 as a general-purpose language for engineering workstations with a single central processor and a few subsidiary input/output processors. He provides for coroutines in a set of low-level facilities and argues that synchronization and communication can and should be built on top of these using an encapsulation facility (in this case, the module).

The reference manual gives an example of a simple process manager encapsulated in a module. This module uses signals for synchronization; communication is done via shared variables. Given coroutines and the module facility, implementing this or alternative designs is essentially an exercise in data structures. Indeed, studying and implement-

ing alternative process managers in this manner serves as a useful project for students.

Implementations of process managers appear in a number of publications, for example Ford and Wiener [Ford85] and King [King88]. Since these are not part of the language [BSI89], but rather programs in it, they are not described in detail. Examples are included in the support materials; here we discuss only the coroutine primitives described in the reference manual.

In some of his writings, Wirth refers to a coroutine by the name *process*. To avoid confusion with our more general use of the term *process*, we prefer to use *coroutine* here; the teacher or student must be careful in understanding whether a given Modula-2 reference using the term *process* means a coroutine or something more general. Usually the meaning is clear from context.

5.1.1. Process Creation and Activation

A coroutine is created by a call to the system operation NEWCOROUTINE (all uppercase; Modula-2 is case-sensitive), whose input arguments are the name of a parameterless procedure and a pointer to a workspace. The output of NEWCOROUTINE is a pointer to a new coroutine. The same procedure can in this way be associated with multiple coroutines. (Older implementations of Modula-2 call this system operation NEWPROCESS.)

Modula-2 provides an operation TRANSFER, whose two arguments are coroutine pointers. The operation transfers control from the first coroutine to the second, suspending the first and resuming the second in the state it was in when it yielded control. A transfer serves as an activation operation since the first transfer to a coroutine effectively activates it.

5.1.2. Process Termination

No explicit termination is provided. Indeed if control flows “off the bottom” of a coroutine, in general the entire program is halted.

5.1.3. Process Scheduling

Coroutine transfer can be seen as an explicit processor-scheduling statement; one coroutine yields control of the processor and transfers control to another, explicitly named, coroutine. This is the main scheduling operation provided in the language.

5.1.4. Process Synchronization

No explicit synchronization is provided.

5.1.5. Interprocess Communication

No explicit communication is provided; coroutines are simply allowed to share any data visible to them.

5.1.6. Nondeterministic Constructs

Nondeterminism in a single-cpu system is caused by interrupts, for example from input/output devices. The Modula-2 reference manual [Wirth85] calls for an operation IOTransfer with three arguments: interrupting coroutine, interrupted coroutine, representation of interrupt vector. The IOTransfer statement is written in an interrupt-handling coroutine; when an interrupt occurs, the current coroutine

yields control to the interrupt handler, which returns control after fielding the interrupt.

Since the point at which the current coroutine is interrupted is determined by an unpredictable event (a device interrupt), this statement can be seen as a nondeterministic construct. Wirth describes an interrupt as an unscheduled coroutine transfer.

Wirth recommends that IOTransfer be written only in an interrupt handler which is encapsulated in a module; it is possible to assign a priority to such a module so that the interrupt handler is not, itself, interrupted. This provides a measure of mutual exclusion.

Versions of Modula-2 for multi-user operating systems (UNIX, VMS) typically do not provide IOTransfer operations, as these are typically the private domain of the operating system.

Teaching Considerations

Where to Use the Module

Concurrent programming can be taught most effectively in the context of a programming project. The module's concentration on readily available systems and languages will make the module directly useful to the widest possible audience.

The material provided here can serve as the basis for a comparative course in concurrent programming per se (as is taught by the author at The George Washington University), or as a unit of a course in operating systems or real-time and distributed systems design.

Objectives

The educational objectives for courses that include material from this module can vary greatly, depending on the kind of course and the module material selected. At the highest level, an objective is that the students learn to examine the concurrent programming capabilities of a programming language by applying the conceptual structure introduced in the module. Students should also be able to distinguish the various support mechanisms for concurrent programming that are provided by a computer system's underlying hardware, its operating system, the features of the various programming languages, and the libraries written in those languages.

In a course that includes a substantial programming project (as mentioned above), more ambitious objectives are possible. Two of the most important are that the students learn to recognize when and how concurrency can be used in the solution of a software design problem and that they become proficient in writing simple concurrent programs in at least one programming language.

If the module is used in support of an operating systems course, the students should understand how different systems can provide very different kinds of concurrent programming support. They should also understand the machine instructions or other hardware features that exist on modern computers in order to support concurrent programming.

Prerequisite Knowledge

This module is designed to follow the module *Concepts of Concurrent Programming*, SEI-CM-24; it is assumed that the reader of this module is familiar with the material introduced there. That module also includes a comprehensive glossary.

Students in a course that includes the material in this module should have basic programming skills, including knowledge of data structures, abstract data types, and subprogram parameter-passing mechanisms. For the operating systems section of the module, familiarity with fundamental concepts of operating systems is desirable.

Example Programs

The program library supplied as supporting material for the module contains examples of programs in five languages. The first example is an implementation of the famous Dining Philosophers problem first stated by Dijkstra [Dijkstra71]. In this metaphorical statement of deadlock and resource allocation problems, five philosophers sit around a circular table, in the center of which is a infinitely large bowl of Chinese food. To the left and right of each philosopher is a single chopstick; each philosopher must try to acquire both chopsticks, eat for awhile, then put down the chopsticks and think for awhile. This cycle repeats for some total number of meals. (Dijkstra's original formulation used spaghetti and forks; we prefer the chopstick setting because most people can eat spaghetti with one fork.) The algorithm for chopstick selection must be chosen carefully; otherwise, if all philosophers grab, say, their left chopsticks and refuse to yield them, all will starve!

The second example is one we have used with repeated success at The George Washington University, namely a "sort race" in which three different sorting methods are activated as processes. Each sort displays its progress in its "window" (usually a single row) on the terminal; mutual exclusion is necessary to protect the screen, which is a writable shared resource. We have found this example interesting and fun– there is a lot of screen activity; the problem being solved is obvious; and the three independent sorts serve as placeholders for any three independent applications contending for the processor and a shared data structure. In our comparative concurrency seminar, students must implement the sort race in the five different languages, starting from modules like sort subroutines, terminal drivers, process managers, etc., supplied by the instructor.

Bibliography

Ackerman82

Ackerman, W.B. "Parallel Processing in Ada." *Computer 15,* 2 (1982), 15-25. Reprinted in [Gehani88].

Andrews81

Andrews, G. R. "Synchronizing Resources." *ACM Trans. Programming Languages and Systems 3,* 4 (1981), 405-430. Reprinted in [Gehani88].

Abstract: A new proposal for synchronization and communication in parallel programs is presented. The proposal synthesizes and extends aspects of procedures, coroutines, critical regions, messages, and monitors. It provides a single notation for parallel programming with or without shared variables and is suited for either shared or distributed memory architectures. The essential new concepts are operations, input statements, and resources. The proposal is illustrated by the solutions of a variety of parallel programming problems; its relation to other parallel programming proposals is also discussed.

Andrews83

Andrews, G. R., and F. B. Schneider. "Concepts and Notations for Concurrent Programming." *Computing Surveys 15,* 1 (1983), 3-43. Reprinted in [Gehani88].

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

Bal89

Bal, H. E., J. G. Steiner, and A. S. Tanenbaum. "Programming Languages for Distributed Computing Systems." *Computing Surveys 21,*3 (1989), 261-322.

This is a particularly thorough and clearly written discussion of programming languages for concurrent programming. It is a useful survey to distribute to students. As is always the case with *Computing Surveys* articles, the bibliography is particularly complete.

Ben-Ari82

Ben-Ari, M. *Principles of Concurrent Programming.* Englewood Cliffs, N. J.: Prentice-Hall, 1982.

This short text is a must for beginning students of concurrent programming. All the important concepts are introduced in a tutorial style that is clear and easy to understand. Formalism is kept to a minimum. A new version of this book was published early in 1990.

Brinch-Hansen75

Brinch-Hansen, P. "The Programming Language Concurrent Pascal." *IEEE Trans. Software Eng. SE-1,* 2 (1975), 199-207. Reprinted in [Gehani88].

Abstract: This paper describes a new programming language for structured programming of computer operating systems. It extends the sequential programming language Pascal with concurrent programming tools processes and monitors. Section I explains these concepts informally by means of pictures illustrating a hierarchical design of a simple spooling system. Section II uses the same example to introduce the language notation. The main contribution of Concurrent Pascal is to extend the monitor concept with an explicit hierarchy of access rights to shared data structures that can be stated in the program text and checked by a compiler.

Brinch-Hansen78

Brinch-Hansen, P. "Distributed Processes: A Concurrent Programming Concept." *Comm. ACM 21,* 11 (1978), 934-941. Reprinted in [Gehani88].

Abstract: A language concept for concurrent processes without common variables is introduced. These processes communicate and synchronize by means of procedure calls and guarded regions. This concept is proposed for real-time applications controlled by microcomputer networks with distributed storage. The paper gives several examples of distributed processes and shows that they include procedures, coroutines, classes, monitors, processes, semaphores, buffers, path expressions, and input/output as special cases.

Bristow79

Bristow, G., C. Drey, B. Edwards, and W. Riddle. "Anomaly Detection in Concurrent Programs." *Proc. Fourth International Conference on Software Engineering.* New York: IEEE, September 1979, 265-273. Reprinted in [Gehani88].

Abstract: An approach to the analysis of concurrent software is discussed. The approach, called anomaly detection, *involves the algorithmic derivation of information concerning potential errors and the subsequent, possibly non-algorithmic determination of whether or not the reported anomalies*

are actual errors. We give overviews of algorithms for detecting data-usage and synchronization anomalies and discuss how this technique may be integrated within a general software development support system.

BSI89

BSI. *Draft British Standard for Programming Language Modula 2: Third Working Draft*. Standard ISO/IEC DP 10514, British Standards Institute, Nov. 1989.

This standard defines a *Processes* and a *Semaphores* module. The semaphore model makes no allowance for the waiting time of processes that are delayed.

Burns85

Burns, A. *Concurrent Programming in Ada.* Cambridge, England: Cambridge University Press, 1985.

This book serves as a nice introduction to the Ada tasking model; sequential Ada is not presented in detail. The second half of the book is especially useful for teachers and students interested in an incisive critique of the tasking model.

Coleman79

Coleman, D., R. M. Gallimore, J. W. Hughes, and M. S. Powell. "An Assessment of Concurrent Pascal." *Software Practice and Experience 9* (1979), 827-837. Reprinted in [Gehani88].

Summary: This paper assesses Concurrent Pascal against its design aims. The language is shown to be suitable for writing reliable non-trivial concurrent applications programs and operating systems. The major weakness of the language is its inability to provide an environment for other Concurrent Pascal programs. A new language construct, group, is proposed to remedy this difficulty.

Cook80

Cook, R. P. "*MOD–A Language for Distributed Computing." *IEEE Trans. Software Eng. SE-6,* 6 (1980), 563-571. Reprinted in [Gehani88].

*Abstract: Distributed programming is characterized by high communications costs and the inability to use shared variables and procedures for interprocessor synchronization and communication. *MOD is a high level language system which attempts to address these problems by creating an environment conducive to efficient and reliable network software construction. Several of the *MOD distributed programming constructs are discussed as well as an interprocessor communication methodology. Examples illustrating these concepts are drawn from the areas of network communication and distributed process synchronization.*

Cornhill87

Cornhill, D., and L. Sha. "Priority Inversion in Ada, or What Should be the Priority of an Ada Server Task?" *Ada Letters vii,* 7 (Nov.-Dec. 1987), 30-32.

Deitel84

Deitel, H. M. *An Introduction to Operating Systems.* Reading, Mass.: Addison-Wesley, 1984.

This book's main strength is in its case studies of major operating systems: UNIX, VMS, MVS, VM. Process management is discussed in reasonable survey depth for each of these.

DoD83

U. S. Department of Defense. *Reference Manual for the Ada Programming Language.* ANSI/MIL-STD 1815A, 1983.

This is the official standard governing the Ada programming language. As language standards go, it is quite well written and not too difficult to navigate. Like all such documents, it is a much better reference than it is a text. It should be made available to anyone studying Ada in any but the most superficial fashion, but is by no means a substitute for a good textbook. Note: a usefully annotated version of this document is now available (and recommended); see [Nyberg89].

Dijkstra68

Dijkstra, E. W. "The Structure of the 'THE' Multiprogramming System." *Comm. ACM 11,* 5 (May 1968), 341-346.

Abstract: A multiprogramming system is described in which all activities are divided over a number of sequential processes. These sequential processes are placed at various hierarchical levels, in each of which one or more independent abstractions have been implemented. The hierarchical structure proved to be vital for the verification of the logical soundness of the design and the correctness of its implementation.

This brief and readable "lessons learned" paper is the earliest widely published reference to the use of semaphores as a synchronization mechanism. It is recommended to both teachers and students as one of the seminal papers in concurrent programming. Like Dijkstra's "goto statement considered harmful" letter of the same year, this paper is often cited, less often read.

Dijkstra71

Dijkstra, E. W. "Hierarchical Ordering of Sequential Processes." *Acta Informatica 1,* 115-138.

In this paper Dijkstra introduced the famous Dining Philosophers problem.

Dijkstra75

Dijkstra, E. W. "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs." *Comm. ACM 18,* 8 (August 1975), 453-457.

Abstract: So-called "guarded commands" are introduced as a building block for alternative and repetitive constructs that allow nondeterministic program components for which at least the activity evoked, but possibly the final state, is not necessarily uniquely determined by the initial state. For the formal derivation of programs expressed in terms of these constructs, a calculus will be shown.

This very formal paper introducing the guarded command paved the way for the practical language construct generally called the select statement. It is a mathematically dense article, suitable reading for the teacher or student wishing to go to primary sources and comfortable with the formalism of program derivation. It is not recommended for a beginner.

Ford85

Ford, G. A., and R. S. Wiener. *Modula-2: A Software Development Approach.* New York: John Wiley, 1985.

This is a very complete discussion of Modula-2; the section on concurrency is useful in its presentation of alternative process schedulers and their encapsulation in modules.

Gehani84a

Gehani, N., and T. A. Cargill. "Concurrent Programming in the Ada Language: The Polling Bias." *Software: Practice and Experience 14* (1984), 413-427. Reprinted in [Gehani88].

Summary: The rendezvous is an important concept in concurrent programming–two processes need to synchronize, i.e. rendezvous, to exchange information. The Ada programming language is the first programming language to use the rendezvous as the basis of its concurrent programming facilities.

Our experience with rendezvous facilities in the Ada language shows that these facilities lead to and encourage the design of programs that poll. Polling is generally, but not always, undesirable because it is wasteful of system resources.

We illustrate and examine the reasons for polling bias in the Ada language. We give suggestions on how to avoid polling programs, and suggest changes to the rendezvous facilities to eliminate the polling bias. The ramifications of these changes to the implementation of the Ada language are also discussed.

Although we have focused on the rendezvous facilities in the Ada language our analysis is also applicable to other languages. A polling bias can occur in any concurrent programming language based on the rendezvous mechanism if it does not provide appropriate facilities.

Gehani84b

Gehani, N. H. "Broadcasting Sequential Processes (BSP)." *IEEE Trans. Software Eng. SE-10*, 4 (1984), 343-351. Reprinted in [Gehani88].

Communication in a broadcast protocol multiprocessor (BPM) is inherently different from that in distributed systems formed by explicit links between processors. A message broadcast by a processor in a BPM is received directly by all other processors in the network instead of being restricted to only one processor. Broadcasting is an inexpensive way of communicating with a large number of processors on a BPM. In this paper I will describe a new approach to user-level distributed programming called Broadcast programming*, i.e., distributed programs written as cooperating broadcasting sequential processes (BSP). Existing concurrent programming languages do not provide facilities to exploit the broadcast capability of a BPM. The idea of distributed programs written as BSP is tailored to exploiting a BPM architecture but is not restricted to such an architecture–however, implementation of the broadcast capability may not be as efficient on other architectures. I will illustrate the utility and convenience of broadcast programming with many examples. These examples will also be used to explore the suitability and advantages of BSP and to determine appropriate facilities for BSP.*

Gehani84c

Gehani, N. *Ada Concurrent Programming.* Englewood Cliffs, N.J.: Prentice-Hall, 1984.

This book contains a useful discussion of the Ada tasking model, and presents a number of good examples. The sequential constructs of Ada are surveyed briefly but not discussed in detail. This is an excellent text on Ada tasking; relatively little attention is paid to the problems of designing systems with concurrent programs.

Gehani88

Gehani, N., and A. D. McGettrick, eds. *Concurrent Programming.* Reading, Mass.: Addison-Wesley, 1988.

This collection of 24 papers is a very important addition to the literature of concurrent programming; most of the seminal papers on concurrent programming models, notations, and languages are included.

Gehani89

Gehani, N., and W. D. Roome. *The Concurrent C Programming Language.* Summit, N.J.: Silicon Press, 1989.

This is the definitive reference to the Concurrent C language. It contains not only a description of the language but many interesting concurrent programming examples. A knowledge of C is assumed; only the concurrency-related extensions are presented in syntactic detail.

Hoare74

Hoare, C. A. R. "Monitors: An Operating System Structuring Concept." *Comm. ACM 17*, 10 (1974), 549-557. Reprinted in [Gehani88].

Abstract: This paper develops Brinch-Hansen's concept of a monitor as a method of structuring an operating system. It introduces a form of synchronization, describes a possible method of implementation in terms of semaphores and gives a subtle proof rule. Illustrative examples include a single resource scheduler, a bounded buffer, an alarm clock, a buffer pool, a disk head optimizer, and a version of the problem of readers and writers.

Hoare78

Hoare, C. A. R. "Communicating Sequential Processes." *Comm. ACM 21*, 8 (1978), 666-677. Reprinted in [Gehani88]

Abstract: This paper suggests that input and output are basic primitives of programming and that parallel composition of communicating sequential processes is a fundamental program structuring method. When combined with a development of Dijkstra's guarded command, these concepts are surprisingly versatile. Their use is illustrated by sample solutions of a variety of familiar programming languages.

Ichbiah79

Ichbiah, J., et al. *Rationale for the Design of the Ada Programming Language.* ACM *SIGPLAN Notices, 14*, 6, June 1979, part B.

This rationale was published along with the preliminary Language Reference Manual, and an earlier version was used as part of the submission to the language design competition sponsored by DoD, which resulted in Ada being selected. This version of the rationale is of particular interest to students and teachers of languages in general and Ada in particular, as a means of comparison with the current rationale [Ichbiah86]. While the broad lines of the language did not change between the preliminary version and the final standard adopted in 1983, the reader of both documents is struck by the number of changes in the direction of uniformity and simplicity. Comparing the two tasking models is especially interesting for students and teachers of concurrent programming.

Ichbiah86

Ichbiah, J, *et al. Rationale for the Design of the Ada Programming Language.* Minneapolis, Minnesota: Honeywell Systems and Research Center, 1986. Available from the U.S. Government National Technical Information Service, and commercially through Silicon Press.

...The original goal was both motivational and defensive; a major concern was implementability...especially since there were no compilers then in existence...The present goal is thus now more inspirational: to give the reader a feel for the spirit of the language, the motives behind the key features and to create the basis for understanding how they fit together both globally as viewed from the outside and in detail as viewed from the inside; above all

to impart an appreciation of the main architectural lines of the language and its overall philosophy. [from the Introduction and Preface]

This is the rationale document corresponding to the current Ada standard [DoD83]. For students of Ada and programming languages in general, this is a useful resource in explaining the "why's and wherefore's" of the language features. The chapter on tasking provides particularly good insight into the design philosophy. Comparisons with alternative choices of language structures appear frequently in the document.

IEEE88

Institute of Electrical and Electronics Engineers. *IEEE Standard Portable Operating System Interface for Computer Environments (POSIX).* IEEE Std. 1003.1-1988.

This is the defining document for the POSIX Standard, that is, the IEEE standard for the C programmer's interface to UNIX-like systems. The rationale, supplied as an addendum to the standard, points out carefully (p. 175) that POSIX is not a completely faithful rendering of any existing UNIX system, but rather a good approximation. "The standard is specifically not a codification of a particular vendor's product. It is like the UNIX system, but it is not identical to it. The word UNIX is not used in the standard proper both for that reason, and because it is a trademark of a particular vendor...The Working Group [the group responsible for writing the standard] wished to make less work for developers, not more. However, because every known historical implementation will have to change at least slightly to conform, some applications will have to change. This Rationale points out the major places where the standard implies such changes."

The UNIX textbook literature has focused primarily on the end-user's interface to UNIX; discussion in any detail on the programmer's interface is rather hard to find. For those interested in the C interface to UNIX system services, this is therefore a very useful document, and is reasonably clearly written. The rationale makes interesting reading, as it explains the design choices made in crafting the standard. Like any standard, though, it is not for beginners.

Kernighan84

Kernighan, B. W., and R. Pike. *The UNIX Programming Environment.* Englewood Cliffs, N. J.: Prentice-Hall, 1984.

This book introduces UNIX at the programmer's level, showing how to write shell scripts, use the translator-writing tools lex and yacc, and so on. Chapter 7, "System Calls," includes a discussion on processes. It is the only text we have located that shows how processes work; even here, the discussion is rather thin and no really nontrivial examples are given.

Kieburtz79

Kieburtz, R. B., and A. Silberschatz. "Comments on 'Communicating Sequential Processes.'" *ACM Trans. Prog. Lang. and Sys. 1,* 2 (1979), 218-225. Reprinted in [Gehani88].

Abstract: In his recent paper, "Communicating Sequential Processes" (Comm. ACM 21, 8 (Aug. 1978), 666-677), C.A.R. Hoare outlines a programming language notation for interprocess communication in which processes are synchronized by the messages they exchange. The notation carries with it certain implications for the synchronization protocols required in a message transfer. These are not at all obvious and are made explicit here. An alternative convention is suggested in which communication and synchronization are partially uncoupled from one another.

King88

King, K. N. *Modula-2: a Complete Guide.* Lexington, Mass.: D. C. Heath, 1988.

As the title suggests, this is a complete guide to the Modula-2 programming language. The chapters on coroutines and process schedulers are especially relevant to students of concurrent programming. Concurrency is shown as a tool for both abstraction and real-time system development. Alternative process schedulers are discussed.

Lampson80

Lampson, B. W., and D. D. Redell. "Experience with Processes and Monitors in Mesa." *Comm. ACM 23,* 2 (1980), 105-117. Reprinted in [Gehani88].

Abstract: The use of monitors for describing concurrency has been much discussed in the literature. When monitors are used in real systems of any size, however, a number of problems arise which have not been adequately dealt with: the semantics of nested monitor calls; the various ways of defining the meaning of wait*; priority scheduling; handling of timeouts, aborts and other exceptional conditions; interactions with process creation and destruction; monitoring large numbers of small objects. These problems are addressed by the facilities described here for concurrent programming in Mesa. Experience with several substantial applications gives us some confidence in the validity of our solutions.*

Liskov83

Liskov, B., and R. Scheifler. "Guardians and Actions: Linguistic Support for Robust, Distributed Programs." *ACM Trans. Prog. Lang. and Sys. 5,* 3 (1983), 381-404. Reprinted in [Gehani88].

Abstract: An overview is presented of an integrated programming language and system designed to support the construction and maintenance of distributed programs: programs in which modules reside and execute at communicating, but geographically distinct, nodes. The language is intended to support a class of applications concerned with the manipulation and preservation of long-lived, on-line, distributed data. The language addresses the writing of robust programs that survive hardware failures without loss of distributed information and that provide highly concurrent access to that information while preserving its consistency. Several new linguistic constructs are provided; among them are atomic actions, and models called guardians that survive node failures.

Liskov86

Liskov, B., M. Herlihy, and L. Gilbert. "Limitations of Synchronous Communication with Static Process Structure in Languages for Distributed Computing." *Proc. 13th ACM Symposium on Principles of Programming Languages.* New York: ACM, January 1986. Reprinted in [Gehani88].

Abstract: Modules in a distributed program are active, communicating entities. A language for distributed programs must choose a set of communication primitives and a structure for processes. This paper examines one possible choice: synchronous communication primitives (such as rendezvous or remote procedure call) in combination with modules that encompass a fixed number of processes (such as Ada tasks or UNIX processes). An analysis of the concurrency requirements of distributed programs suggests that this combination imposes complex and indirect solutions to common problems and thus is poorly suited for applications such as distributed programs in which concurrency is important. To provide adequate expressive power, a language for distributed programs should abandon either synchronous communication primitives or the static process structure.

Maekawa87

Maekawa, M., A. E. Oldehoeft, and R. R. Oldehoeft. *Operating Systems: Advanced Concepts.* Menlo Park, Calif.: Benjamin Cummings, 1987.

This is a second-level text in operating systems, focusing on recent results and concepts. Chapter 3 of the book is a nice survey of language mechanisms for concurrency; Chapters 2 and 4 cover synchronization and deadlock issues, respectively. The second half of the book covers distributed systems: distributed concurrency control and deadlock problems, network structures, etc. Recommended for students with prior background in operating systems; not recommended for beginners.

Morris79

Morris, J. M. "A starvation-free solution to the mutual exclusion problem." *Information Processing Letters 8* (1979), 76-80.

Nyberg89

Nyberg, K. *The Annotated Ada Reference Manual.* Vienna, Virginia: Grebyn Corporation, 1989.

This book contains the full text of ANSI/MIL-STD-1815A-1983, the "Reference Manual for the Ada Programming Language" (LRM), with inline annotations derived from the Ada Issues (AIs—sometimes better known as the Ada Commentaries). The version of the AIs used in deriving the annotations in this manual were those available as of the 24th of June, 1989.

The primary purpose of the annotations is to provide greater understanding and insight into the LRM by including relevant text from the AIs in a manner in which they are included inline as they apply. [from the Preliminary section]

This is a good source for teachers and students of Ada, as it incorporates a fair amount of officially sanctioned commentary material into the text of the language reference manual. The commentaries represent the official position of the Ada Joint Program Office on many questions of unclarity or ambiguity, raised over the years by Ada developers and users. In the area of concurrent programming, the Chapter 9 annotations are often very helpful in understanding some of the interpretations, ambiguities, and open questions in the Ada tasking model.

Pountain87

Pountain, D., and D. May. *A Tutorial Introduction to occam Programming.* Oxford, U.K.: BSP Professional Books, 1987.

This is a textbook on the occam programming language. David May was responsible for designing the occam language and developing the Transputer architecture for which occam was originally conceived.

Quarterman85

Quarterman, J. S., A. Silberschatz, and J. L. Peterson. "4.2BSD and 4.3BSD as Examples of the UNIX System." *Computing Surveys 17,* 4 (1985), 379-418.

Abstract: This paper presents an in-depth examination of the 4.2 Berkeley Software Distribution, Virtual VAX-11 Version (4.2BSD), which is a version of the UNIX Time-Sharing System. There are notes throughout on 4.3BSD, the forthcoming system from the University of California at Berkeley. We trace the historical development of the UNIX system from its conception in 1969 until today, and describe the design principles that have guided this development. We then present the internal data structures and algorithms used by the kernel to support user interface. In particular, we describe process management, memory management, the file system, the I/O system, and communications. These are treated in as much detail as the UNIX licenses will allow. We conclude with a brief description of the user interface and a set of bibliographic notes.

Similar to the UNIX chapter in [Silberschatz88], this is a readable survey of the facilities in the Berkeley implementation of UNIX. The discussion of processes and interprocess communication is particularly well-summarized and relevant to this module.

Roberts81

Roberts, E. S., A. Evans, Jr., C. R. Morgan, and E. M. Clarke. "Task Management in Ada." *Software: Practice and Experience 11* (1981), 1019-1051. Reprinted in [Gehani88].

Summary: As the cost of processor hardware declines, multiprocessor architectures becomes increasingly cost-effective and represent an important area for future research. In order to exploit the full potential of multiprocessors, however, it is necessary to understand how to design software which can make effective use of the available parallelism. This paper considers the impact of multiprocessor architecture on the design of high-level programming languages and, in particular, evaluates the language Ada in the light of the special requirements of real-time multiprocessor systems. We conclude that Ada does not, as currently designed, meet the needs for real-time embedded systems.

Rochkind85

Rochkind, M. *Advanced UNIX Programming.* Englewood Cliffs, N. J.: Prentice-Hall, 1985.

This book is devoted mainly to advanced file handling and concurrent programming with the UNIX operating system. It is the only source we have found in the textbook literature which gives concurrent programming examples detailed enough to show the techniques and the problems effectively. The book is not for the faint-hearted or the beginning student. The examples are involved, often tricky, and not always portable. An operating-systems-oriented instructor can use them selectively to teach the "how" of concurrent programming in UNIX using C. The intricacy of these examples makes a very good case for concurrent programming at the language level!

Roubine80

Roubine, O., and J.-C. Heliard. "Parallel Processing in Ada." in McKeag, R. M., and A. M. Macnaghten, *On the Construction of Programs.* Cambridge, U.K.: Cambridge University Press, 1980, 193-212. Reprinted in [Gehani88].

Schoening86

Schoening, C. "Concurrent Programming in Co-Pascal." *Computer Language* (Sept. 1986), 33-37.

Shumate88

Shumate, K. *Understanding Concurrency in Ada.* New York, N.Y.: McGraw-Hill, 1988.

This book presents the Ada tasking model in considerable detail, and includes five nontrivial case studies of real-time programs using tasks.

Silberschatz88

Silberschatz, A., and J. L. Peterson. *Operating System Concepts, Alternate Edition.* Reading, Mass.: Addison-Wesley, 1988.

This is a new edition of a very popular text on operating systems. "Alternate edition" refers to the fact that discussions of concurrency are moved to the front of the book and emphasized. Chapter 12 is a quite readable survey of the UNIX operating system. A similar description of UNIX is available in [Quarterman85].

Stotts82

Stotts, P. D. "A Comparative Survey of Concurrent Programming Languages." *SIGPLAN Notices 17*, 10 (1982), 76-87. Reprinted in [Gehani88].

Abstract: In the past decade, development of a body of theory for parallel computation has fostered research into the design of programming languages intended to facilitate expression of concurrent computations. Referred to in this report as concurrent languages, or parallel languages, their number is not nearly as great as that of strictly sequential languages.

Many, however, are available for academic study and general use, and their number is increasing yearly.

Stroustrup86

Stroustrup, B. *The C++ Programming Language.* Englewood Cliffs, N. J.: Prentice-Hall, 1986.

Tanenbaum87

Tanenbaum, A. S. *Operating Systems Design and Implementation.* Englewood Cliffs, N. J.: Prentice-Hall, 1987.

This is a useful practical text on operating systems. The Minix operating system, a variant of UNIX, is used as the design example throughout.

Taylor83

Taylor, R. N. "A General-Purpose Algorithm for Analyzing Concurrent Programs." *Comm. ACM 26,* 5 (1983), 362-376. Reprinted in [Gehani88].

Abstract: Developing and verifying concurrent programs presents several problems. A static analysis algorithm is presented here that addresses the following problems: how processes are synchronized, what determines when programs are run in parallel, and how errors are detected in the synchronization structure. Though the research focuses on Ada, the results can be applied to other concurrent programming languages such as CSP.

Wegner83

Wegner, P., and S. A. Smolka. "Processes, Tasks and Monitors: A Comparative Study of Concurrent Programming Primitives." *IEEE Trans. Software Eng. SE-9,* 4 (1983), 446-462. Reprinted in [Gehani88].

Abstract: Three notations for concurrent programming are compared, namely CSP, Ada, and monitors. CSP is an experimental language for exploring structuring concepts in concurrent programming. Ada is a general-purpose language with concurrent programming facilities. Monitors are a construct for managing access by concurrent to shared resources. We start by comparing "low-level" communication, synchronization, and nondeterminism in CSP and Ada and then examine "higher-level" module interface properties of Ada tasks and monitors.

Similarities between CSP and Ada include use of "cobegin" construct for nested process initiation and the "rendezvous" mechanism for synchronization. Differences include the mechanisms for task naming and nondeterminism. One-way (procedure-style) naming of called tasks by calling tasks in Ada is more flexible than the two way naming CSP. The general-purpose nondeterminism of guarded in CSP is cleaner than the special-purpose nondeterminism of the select statement in Ada.

Monitors and tasks are two different mechanisms for achieving serial access to shared resources by concurrently callable procedures. Both rely on queues to achieve serialization, but calls on monitor procedures are scheduled on a single monitor queue while task entry calls are scheduled on

separate queues associated with each entry name. Monitors are passive modules which are activated by being called, while tasks are active modules that execute independently of their callers. Monitor procedures represent multiple threads of control each of which may be suspended and later resumed, while tasks have just a single thread of control. The attempt to map a monitor version of a shortest job scheduler into Ada yields interesting insights into the limitations of Ada mechanisms for synchronization, and suggests that Ada packages may be appropriate than tasks as a user interface for concurrent computation.

Wetherell80

Wetherell, C. "Design Considerations for Array Processing Languages." *Software: Practice and Experience 10* (1980), 265-271. Reprinted in [Gehani88].

Summary: The Department of Energy (DoE) has a long history of large-scale scientific calculation on the most advanced "number crunching" computers. Recently, an effort to improve communications and software sharing among DoE laboratories has been underway. One result of this sharing is a project to design and implement a common language. That language turns out to be FORTRAN 77 significantly extended with new data structures, control structures and array processing. The data used to design the array processing feature is surprising and likely to be of use to others working in scientific language design; it is reported here so that others may profit from DoE's experience.

Wirth84

Wirth, N. *Schemes for Multiprogramming and Their Implementation in Modula-2.* Zurich, Switzerland: ETH, 1984. Reprinted in [Gehani88].

Two sets of primitive operators for communication among concurrent processes are presented. They characterize the schemes of communication via shared variables *and signals for synchronization and by* message passing *through channels. Their use is demonstrated by simple and typical examples, and their implementation is described in terms of Modula-2. Both implementations are based on coroutines for a single-processor computer (Lilith). The primitives for coroutine handling are also presented as a Modula-2 module, demonstrating the language's low-level facilities. A variant of the module for signals is adapted to the needs of discrete event simulation.*

These modules, whose implementation is brief and efficient, demonstrate that, although Modula-2 lacks a specific construct for multiprogramming, such facilities are easily expressible and that a programmer can choose a suitable scheme by selecting the appropriate low-level (library) module. The conclusion is that, if a language offers low-level facilities and an encapsulation construct, it need not offer specific features for multiprogramming. To the contrary, this might hamper the programmer in his search for an appropriate and effective solution.

Wirth85

Wirth, N. *Programming in Modula-2, 3rd edition.* Springer-Verlag, 1985.

This is the definitive reference on Modula-2. Wirth's writing style and the book's typography make it unsuitable for use as a textbook on the language; also, subtle changes from edition to edition have created difficulties for compiler implementors. The book's usefulness is in revealing Wirth's ideas and language philosophy, which have not changed across editions; the book therefore makes good background reading for teachers and students interested in a language designer's rationale and motivation.