On implementing semaphores with sets

J. L. Keedy*, K. Ramamohanarao and J. Rosenberg

Department of Computer Science, Monash University, Clayton, Victoria 3168, Australia

It is proposed that semaphores should in some circumstances be extended by associating with each semaphore, in addition to the usual integer, a set (bit string). There are two separate uses for this set, which are considered separately and which can be implemented independently of each other. The first, the available resources set, has a bit identifying each resource controlled by the semaphore, which will indicate when the resource is free. When no resources are free the set may then be used to identify those processes waiting on a resource. The advantages and disadvantages of both sets are discussed, including the possibility of eliminating MUTEX semaphores from situations such as producer/consumer activities, and the possibility of entirely 'automating' (i.e. controlling by hardware semaphore instructions) the synchronisation and scheduling of processes.

(Received February 1977)

1. Introduction

This paper is concerned primarily with the efficient implementation of semaphores, in particular the *general semaphore* (Dijkstra, 1968a; 1968b; 1972). The method proposed is to supplement the semaphore integer with a *set* (i.e. a bit string in which each bit position represents the absence (0) or presence (1) of a member of the set). The set will be referred to as SEMSET, and may be thought of as occupying one or more words adjacent to the integer SEMINT.

Just as SEMINT is usually interpreted as holding a count of available resources (when positive) and a count of processes waiting to acquire a resource (when negative), so SEMSET potentially represents the corresponding set of available resources or the set of waiting processes.† The arguments for and against implementing these two sets are unrelated, and are discussed separately. The result is that in a particular environment it might be preferable it implement neither, one or both sets, as we shall see.

2. Typical semaphore implementation

The standard method of implementing the semaphore primitives P and V is as operating system routines which execute non-interruptably in the innermost level (or *Kernel*) of the system, usually as part of the process scheduler. As Parnas (1975) recently pointed out, the code of such primitives should be small and quickly executed, and one should avoid the temptation of building higher level abstractions of semaphores into this level of the system. As the discussion of the waiting process set unfolds, it will be observed that in favourable environments the size and complexity of the non-interruptible Kernel code required to achieve P and V operations reduces considerably, and can even disappear entirely.

The means of achieving these gains is by suitable hardware (or microcoded)‡ instructions. Some computers already provide basic instructions which indivisibly increment and decrement a main memory operand. For example, the ICL2900 Series (Keedy, 1977) implements the following pair of indivisible operations on (semaphore) integer operands:

INCT (corresponding to P) increments a main memory location and sets a condition code on the result.§

TDEC (corresponding to V) sets a condition code to indicate the status of a main memory location then decrements it.

Using these instructions P and V can become in-line macros in the users' code (**Figs. 1 and 2**) provided that the Kernel supports a commutative event system.¶

Note that in this implementation of P and V the processor need not be non-interruptible except whilst executing INCT/TDEC in the hardware, or within the event system. But the main advantages of this implementation are that (*a*) in a considerable number of cases the event system will not be called, thus eliminated the overheads associated with supervisor calls; and (*b*) the semaphore integer resides in pageable memory allocated to the user rather than in locked-down Kernel memory.

3. Resource sets

3.1 *Dijkstra's General Semaphore*

The elegance of Dijkstra's general semaphore as a mechanism for handling the exclusive allocation of sets of equivalent resources, such as the use of a bounded buffer by a co-operating community of producers and consumers (**Fig. 3**), especially when compared with previous synchronising attempts (cf. Dijkstra, 1968a), has lulled us into accepting a major inconvenience: the intrusion of an additional binary semaphore

```
INCT (SEM);
IF (CONDITION CODE INDICATES NO FREE
   RESOURCES) THEN suspend on event (semevent);
```

Fig. 1 The *P* operation as an in-line macro

```
TDEC (SEM);
IF (CONDITION CODE INDICATES WAITING
   PROCESSES) THEN cause event (semevent);
```

Fig. 2 The *V* operation as an in-line macro

*At Technische Hochschule Darmstadt, Institut für Praktische Informatik, Fachbereich 20, 61 Darmstadt, Steubenplatz 12, West Germany, until May 1979.

†Note that SEMINT can theoretically be replaced by a Boolean variable indicating which set is represented, but in practice the integer has several uses (e.g. it contains a record of the current set size).

‡If these means are not available then non-interruptible software routines (typically in the Kernel) can be made to simulate the hardware semaphore set operations.

§In ICL2900 semaphores, positive values represent counts of waiting processes, and negative values count available resources.

¶In a commutative event system, cause event (x) + suspend on event (x) = suspend on event (x) + cause event (x), i.e. the queue for an event channel (unless empty) contains either a record of causes or a record of interested processes; each event releases one waiting process, and the order in which they occur is immaterial. For the problems associated with non-commutative event systems, see Dijkstra (1972), p. 81.

Producer protocol	*Consumer protocol*
P (EMPTY);	*P* (FULL);
P (MUTEX);	*P* (MUTEX);
add a portion to the buffer;	take a portion from the buffer;
V (FULL);	*V* (EMPTY);
V (MUTEX);	*V* (MUTEX);

Fig. 3 Dijkstra's solution to the bounded buffer problem

P (PRINTER);
P (MUTEX);
use the printer;
V (PRINTER);
V (MUTEX);

Fig. 4 An inadequate solution for allocating printers

P (PRINTER);
P (MUTEX);
find which printer;
V (MUTEX);
use the printer;
P (MUTEX);
note which printer is released;
V (PRINTER);
V (MUTEX);

Fig. 5 A correct solution for allocating printers

Producer protocol	*Consumer protocol*
RP (EMPTY, *x*);	*RP* (FULL, *y*);
add portion to buffer area *x*;	take portion from buffer area *y*;
RV (FULL, *x*);	*RV* (EMPTY, *y*);

Fig. 6 The bounded buffer solution using resource sets

RP (PRINTER, *Z*);
user printer *Z*;
RV (PRINTER, *Z*);

Fig. 7 Allocating printers using resource sets

(usually known as MUTEX, mutual exclusion). It is inconvenient because its use is not intuitively obvious in most cases and it is therefore likely to be a source of errors. For example, an unsuspecting programmer might 'deduce' from Fig. 3 that a good solution for allocating printers to processes is that shown in **Fig. 4**, and he will be amazed that at any instance in time only one printer is active. Hopefully he will modify his algorithm as illustrated by **Fig. 5**. He might also sequence the *P* operations incorrectly. (Fortunately the sequencing of *V* operations in the example provided is irrelevant, but will he realise this?)

The use of MUTEX in these algorithms is inescapable unless the semaphore operations are upgraded such that (*a*) *P* returns an identifier indicating *which* resource has been allocated (rather than merely *that* a resource has been allocated, making it necessary to have exclusive access to state variables in order to determine the identity of the resource) and (*b*) *V* receives the identity of the freed resource. Given a new set of semaphore macros *RP* (semaphore identity, resource acquired) and *RV* (semaphore identity, resource freed) then the MUTEX semaphore can be eliminated (**Figs. 6 and 7**), with the added bonus, in the producer/consumer environment, of increased concurrency (which could have been achieved otherwise by additional use of MUTEX). However, such an extension to *P* and *V* operations can only be justified if an efficient implementation is possible.

3.2 *Implementation of resource sets*

At this stage the use of a commutative event system is retained but into the hardware semaphore instructions is introduced manipulation of available resources set (in SEMSET) whenever SEMINT $\geqslant 0$. **Fig. 8** provides logical flowcharts of the new hardware instructions *RSETP* and *RSETV* (corresponding to *RP* and *RV*). The resource identity parameter is represented by the symbol *R* (a dedicated register, a general purpose register or a memory address) which holds the integer identity of a resource (corresponding to the integer value of a bit position in SEMSET, numbered from left to right or from right to left, as proves convenient).

Typically, SEMINT is initialised to the number of available resources (as with the general semaphore) and in SEMSET a bit corresponding to the identity of each available resource is set. **Figs. 9 and 10** show how the macros *RP* and *RV* are implemented, and at this stage it is assumed that a message can be passed with each event, so that the value of *R* can be passed when *RSETP* cannot immediately allocate a resource.*

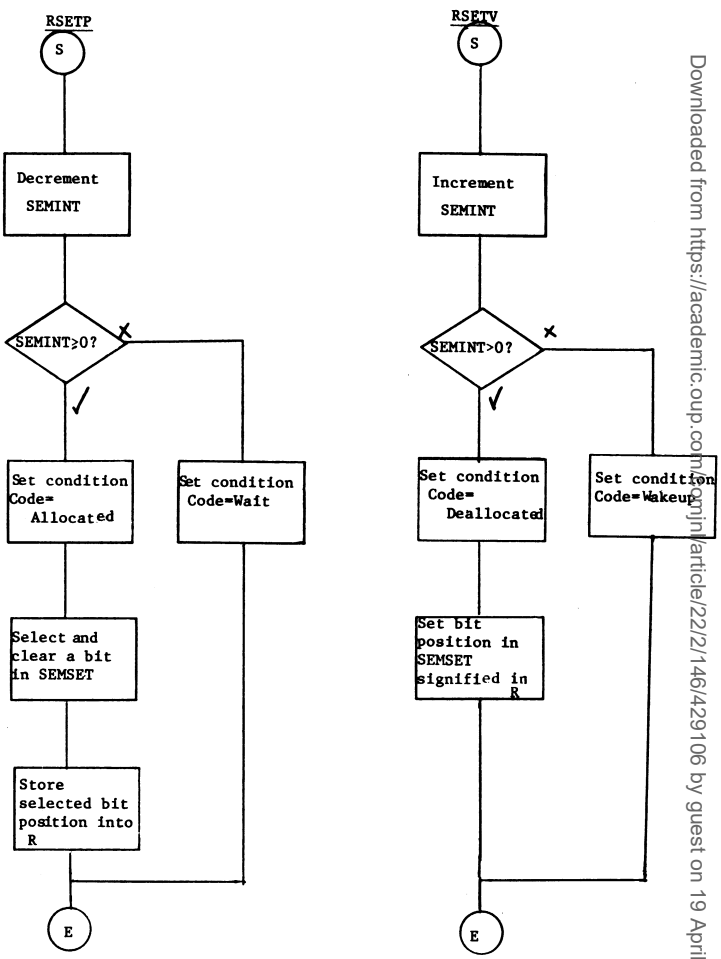

Fig. 8 Set semaphore operations for resource allocation

RSETP (SEMINT, *R*);
IF (CONDITION CODE = WAIT)
THEN suspend on event (semevent, *R*);

Fig. 9 The resource allocating *P* macro

RSETV (SEMINT, *R*);
IF (CONDITION CODE = WAKEUP)
THEN cause event (semevent, *R*);

Fig. 10 The resource releasing *V* macro

*In practice, the association of a message with an event is useful independently of resource set semaphores, since (*a*) it allows passing of status information associated with hardware interrupt events (e.g I/0 terminations), and (*b*) it provides users with a message passing facility. It is implemented in the ICL 2900 event system.

3.3 *Advantages of implementing resource sets*

The most important advantage of implementing resource set semaphores is the improvement in protocols for allocating resources. The elimination of MUTEX makes the resource allocating protocols obvious and natural, thus rendering them considerably less error-prone.

Eliminating MUTEX also results in greater system efficiency. First, a semaphore and its corresponding queue of waiting processes disappears entirely. Second, the associated supervisor call overheads disappear. Third, the concurrency of the system will potentially increase (e.g. with producers and consumers), because it is often not worth the overheads of increasing the use of MUTEX to reduce by a few instructions the time that another process is locked out.

Finally, how do resource set semaphores affect the binary semaphore? Apart from the elimination of MUTEX-type binary semaphores, there are 'more genuine' cases where a single resource is allocated, e.g. in a single printer system. In such cases resource sets are not necessary, in that the identity of the resource can be implied. But the careful programmer will still prefer the *RP* and *RV* operations, just in case his installation buys a second printer! Even in the case of allocating entries in a global table, the *RP* and *RV* operations can offer considerable advantages, if the entries can be considered as equivalent resources (but a MUTEX semaphore may prove to be useful for *accessing*, rather than allocating, table entries, either on a one-per-table or one-per-entry basis).

3.4 *Limitations of resource sets*

A potential limitation of resource sets is the number of bits available in SEMSET. For most practical purposes a single computer word will suffice, because typically where large sets of resources are being controlled, a more sophisticated (i.e. software implemented) algorithm is required.† It is convenient at this point to underline the fact that resource set semaphores are primitive operations, and not a general panacea for all resource allocation problems. Like general semaphores, resource set semaphores do not solve deadlocks, nor resource allocation by priority of waiting processes, nor can non-equivalent resources be scheduled together.

A further limitation is discussed (and resolved) in the next section.

3.5 *The cyclic resource set semaphore*

Previous discussion of *RSETP* ignored the criterion used for allocating a free resource. An efficient algorithm might search the bits in SEMSET from left to right (or right to left as appropriate) starting at the first (or last) bit. Unfortunately this has at least two side effects. First it imposes on resources an allocation priority depending on the identity of each resource. Whilst this will usually have a neutral or beneficial effect with software resources such as table entries (always allocating entries in a fixed length table from the top may reduce search times), for printers and other mechanical devices it may have the effect of wearing out printer 0 long before printer 1 or printer 2.‡

Of more importance, the allocation algorithm is likely to produce incorrect results where a single producer creates a significant sequence of outputs to be consumed by a single consumer (e.g. a user program produces print lines to be printed by a printer process).

Both problems are solved by adding a pointer (SEMSETPTR) to the semaphore, which indicates where the next search of SEMSET should begin (i.e. with the bit following that corresponding to the last resource allocated).

†e.g. allocation of cylinders on a disc pack usually requires the capability of allocating as a unit several adjacent cylinders; equally, the problem of allocating page frames of a main memory is primarily a *discard* problem.

‡This is probably also true of most software schedulers.

4. The waiting process set

Unlike the resource set, the waiting process set does not extend the scope of *P* and *V* operations. It does, however, offer possibilities for efficient implementation of semaphores in certain scheduling environments.

4.1 *Implementation of waiting process sets*

Whenever SEMINT < 0, each bit in SEMSET represents a unique process and its bit position in the set corresponds to its process number. Processes appear in the set as a result of suspensions during a *P* operation, and are removed from the set by subsequent *V* operations. Logical flowcharts of the appropriate hardware instructions *WSETP* and *WSETV* are shown in **Fig. 11**. Resource set operations are ignored in order to concentrate attention on waiting process sets, but the symmetry with resource set handling is clear. For the *WSETP* operation an input register *W* (which might be set up by the calling process, or could be an implied system register which holds the integer identity of the current process) is assumed. This is used by the hardware if the calling process is suspended. Correspondingly, the *WSETV* operation may advise the caller that a certain process should be woken up, in the *W* register (which in this case cannot be a system status register, but may be a dedicated or general purpose register or a reference to a main memory location).

Incorporation of these hardware instructions into user level *P*

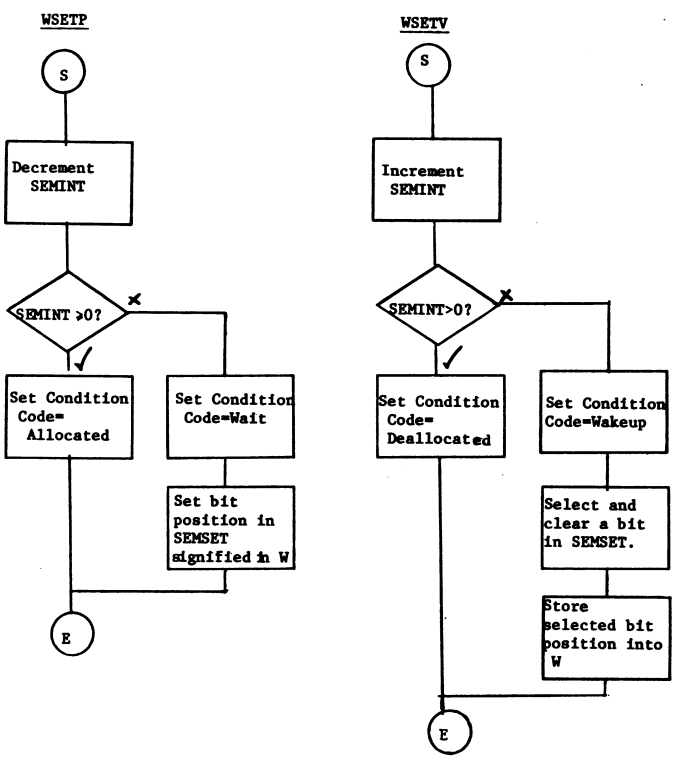

Fig. 11 Set semaphore operations for waiting processes

WSETP (SEMINT, *W*);
IF (CONDITION CODE = WAIT)
THEN suspend (*W*);

Fig. 12 An in-line macro for *P* using waiting process sets

WSETV (SEMINT, *W*);
IF (CONDITION CODE = WAKE UP)
THEN wake up (*W*);

Fig. 13 An in-line macro for *V* using waiting process sets

and *V* macros (**Figs. 12 and 13**) allows us to reduce the previously necessary commutative event system into a straightforward process scheduling system which implements directives to suspend and wake up processes.* It also must be commutative in case an interrupt occurs between *WSETP* and the subsequent suspend call. The appropriate mechanism is easy to implement, being analogous to Dijkstra's one-per-process *private* semaphores (1968b).

4.2 *Selecting a process from the waiting process set*
As with the *RSETP* instruction (Section 3.5), the two selection criteria which can feasibly be implemented are (*a*) to begin all searches at one end of SEMSET, or (*b*) to maintain a cyclic pointer (SEMSETPTR). Neither of these corresponds to the FIFO queueing discipline usually associated with semaphores. Searches of type (*a*) imply a fixed priority of processes, with process numbers directly related to priority. Searches of type (*b*) correspond neither to FIFO nor to priority queueing, but ensure that where process numbers are arbitrarily assigned no process is permanently disadvantaged by its process number, and furthermore that the only queueing condition imposed by Dijkstra (1972), viz. that 'no process will be blocked indefinitely', is fulfilled.

Although the inability of the set implementation to provide FIFO queueing is unfortunate, it should be remembered that according to Dijkstra (1968b) it is undefined which process is removed from the waiting list following a *V* operation.

4.3 *Advantages of implementing waiting process sets*
In addition to the efficiency gains resulting from implementing *P* and *V* as in-line user macros (cf. Section 2), waiting process sets are capable of enabling operating system designers to replace a fairly complex event system by a more trivial process scheduling system. This reduces the size and complexity of the Kernel, and allows the processor to spend less of its time executing non-interruptably. In other words the result will be, in Parnas' terms (1975), a very primitive set of kernel mechanisms.

4.4 *Limitations of waiting process sets*
Apart from the limited criteria available for selecting a process from the waiting process set, the chief limitation of waiting process set semaphores (like normal semaphores) is that they genuinely are primitive operations. Thus, it is not easy to use the hardware or process scheduling primitives to achieve complex synchronising requirements (e.g. to suspend a process on the union of several events, may be a timer interrupt or a normal command or a break-in; or to allow a process to be interrupted (to a nominated procedure) when an event occurs).

4.5 '*Automatic*' *scheduling*
Assuming that either the priority or the cyclic selection algorithm is acceptable, then it may be feasible to 'automate' process scheduling entirely if the process scheduler finds the same criterion acceptable. In this case the *WSETP* and *WSETV* instructions will also maintain on a system wide basis a further set, the set of ready processes, which for convenience is assumed to reside in a system register RP, and uses the same selection criterion as that for selecting a process to wake up on a *WSETV* operation (with a global cyclic pointer if necessary).

Each *WSETP* and *WSETV* instruction (**Fig. 14**) will, if necessary, conclude by selecting a new process from RP and effect the necessary process switch by storing and loading registers, etc. (To do this the instructions will require to know

*If resource sets are also implemented then a simple facility for passing a message must be associated with these Kernel routines.

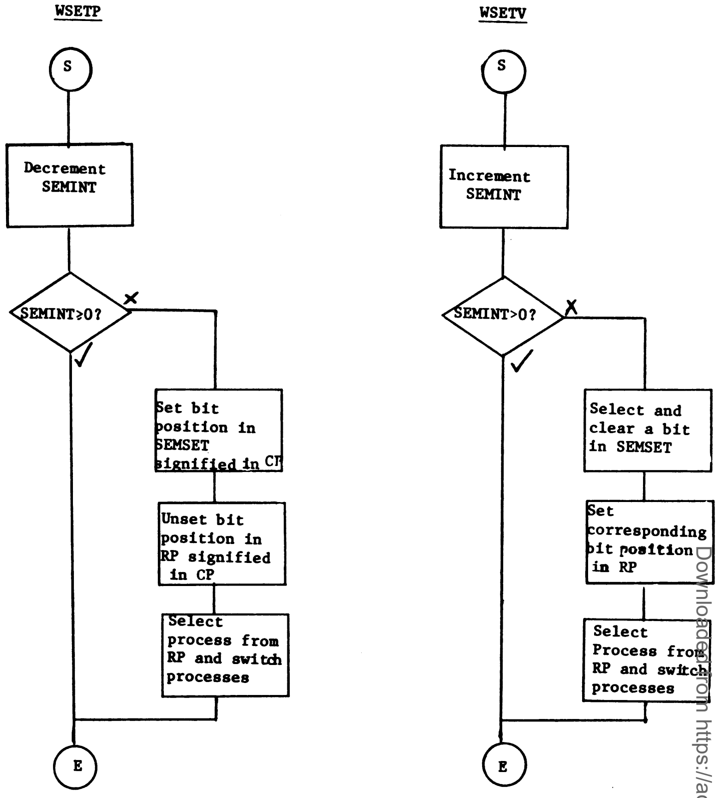

Fig. 14 Automatic scheduling using semaphores

the address of a save area for each process's registers and will maintain a further register CP which holds the integer identity of the current process. Such instructions are quite feasible and may be compared with process switching instructions such as 'move to stack' on the B6700 (Burroughs, 1972). The process selection algorithm must be capable of handling the special case where no processes are ready to run (i.e. where all processes are suspended on *V* operations—including hardware interrupts).

5. Conclusion
Subject to the limitations already mentioned, implementing semaphores with sets of resources and/or waiting processes can achieve considerable gains in the efficiency of a system: by reducing the overheads in calls to the operating system; by reducing the size and complexity of operating system code; and by increasing the potential for parallel processing. Additional advantages of resource sets are that they increase the power of *P* and *V* operations, and simplify their use in the allocation of multiple equivalent resources, whilst at the same time offering an efficient implementation.

Appendix
The ideas expressed in this paper have arisen in the context of the design work for the MONADS operating system, a research project being developed in the Computer Science Department of Monash University using a modified HP2100A computer.

An initial version of the MONADS 'Hardware Kernel' (the innermost operating system level, whose function is to offer an improved 'hardware' interface) has been developed using microcoded resource set semaphores. This limited experience confirms their usefulness and simplicity of use. So far they have given rise to no problems. We have also found that if the SEMINT and SEMSET manipulations in *RSETP* and *RSETV* are implemented as subroutines in the microcode, then the subroutines can also be usefully employed to implement (*a*) standard *P* and *V* semaphore instructions, and (*b*) non-semaphore instructions for searching bit strings and

setting named bits, at virtually no extra cost.

These same subroutines will also be employed to implement the automatic scheduling philosophy described in Section 4.4, in a trial reimplementation of the Hardware Kernel's process scheduler (using priority scheduling). We chose not to implement this in the first version, out of respect for the problems of debugging the microcode, the synchronising design and the code of the Kernel processes simultaneously!

References

BURROUGHS. (1972). The Burroughs B6700 Information Processing Systems Reference Manual, Burroughs Corporation, Detroit, Michigan, 1972.

DIJKSTRA, E. W. (1968a). Cooperating Sequential Processes in *Programming Languages*, ed. F. Genuys, Academic Press, London and New York, 1968.

DIJKSTRA, E. W. (1968b). The Structure of the 'THE'-Multiprogramming System, *CACM*, Vol. 11, No. 5, p. 341.

DIJKSTRA, E. W. (1972). Hierarchical Ordering of Sequential Processes in *Operating System Techniques*, ed. C. A. R. Hoare and R. H. Perrott, Academic Press, London and New York, 1972.

KEEDY, J. L. (1977). An Outline of the ICL2900 Series System Architecture, *Australian Computer Journal*, Vol. 9, No. 2, July, 1977.

PARNAS, D. L. (1975). On a Solution to the Cigarette Smokers' Problem (without conditional statements), *CACM*, Vol. 18, No. 3, p. 181.

Book reviews

Compiler Design Theory, by P. M. Lewis II, D. J. Rosenkrantz and R. E. Stearns, 1976; 647 pages. (*Addison Wesley*, £18·50)

This book is intended as an undergraduate course on compiler design theory. Some teachers may prefer not to base their course on such a formal, and largely automata theory, footing but all should find the book a rich source of 'academically respectable' material which could supplement a more practical treatment of the subject. It is meticulously prepared with many worked and even more unworked examples. The authors' claim that it should be understandable to a wide range of readers is justified, but it is by no means light reading.

Although the approach is consistently and deliberately formal the application of the concepts is demonstrated on the design of an actual compiler. Unfortunately, in the reviewer's opinion, the language chosen for this demonstration, a subset of BASIC, presents little challenge to the compiler writer. A reader might be excused for feeling that the techniques described represent 'sledge hammer to crack a nut'.

The first four chapters give a good and adequate introduction to finite state machines leading up to the design of the lexical analysis stage for the BASIC subset compiler. At this point push down machines are introduced and their application as recognisers and translators of abstract input sequences is considered.

After a diversion in Chapters 6 and 7 to introduce the basic theory of context free grammars, translation grammars and attributed translation grammars, the application of push down machines to top down parsing and translation is developed. This leads in Chapter 10 to the detailed design of the syntax analysis phase of the BASIC subset compiler.

The alternative of bottom up analysis is described in Chapters 11, 12 and 13, again in terms of a push down machine implementation. An alternative (compatible) design for the syntax analysis phase of the BASIC subset compiler is given.

Code generation and optimisation are the subject of the final two chapters of the book. Many compiler writers might feel that this is the area where the main problems lie, but this book devotes only 31 of its 600-odd pages to them. Perhaps the problems are symptomatic of the dearth of formal treatment of the subject.

In conclusion it must be said that this book presents a great deal of compiler theory in a new way and is worthy of a place on any computer scientist's bookshelf. However, if funds are a limiting factor, the book *Principles of Compiler Design* by Aho and Ullmann might be a preferred alternative since it is more broadly based.

D. MORRIS (Manchester)

An Introduction to Programming and Problem Solving with Pascal, by G. Michael Schneider, Steven W. Weingart and David M. Perlman, 1978; 394 pages. (*John Wiley*, £9·20)

This book, like a growing number in recent years, is derived from programming courses which have been taught at university, and is based on the PASCAL programming language. The main aim of the book is to introduce all of the aspects concerned with programming and problem solving, from problem specification to design, implementation, debugging and documentation and maintenance. Secondary aims are to teach what constitutes a good programming style, and to teach the syntax of PASCAL.

Following the introduction, Chapter 2 informally introduces the concepts of algorithms and the basic forms of flow of control. The chapter also discusses the efficiency of algorithms and recursion. Chapters 3 to 8 introduce various aspects of PASCAL, with Chapter 6 devoted to the important topic of debugging and testing programs. Chapter 9, entitled 'Building quality programs', is rightly regarded by the authors as being one of the most important chapters in the book and discusses techniques for developing and managing large 'real world' programs.

The book is well presented and has many illustrative examples. Each chapter is followed by exercises and some solutions are provided. Some people will no doubt find minor faults with the order of presentation or with the emphasis given (or not given) to some topics. However, the book does appear to have been well thought out (probably at the expense of many undergraduates!) and to satisfy its aims. It should certainly be considered as a contender for a recommended textbook for introductory programming courses, particularly those based on PASCAL.

P. A. LEE (Newcastle)

Principles of Compiler Design, by A. V. Aho and J. D. Ullmann, 1977; 604 pages. (*Addison-Wesley*, £15·20)

This book will be valuable as an encyclopaedia of techniques used in the construction of compilers. It deals in depth with the construction of lexical scanners, basic parsing techniques, the mechanical construction of LR, SLR, and LALR parsers (including a useful exposition of the treatment of ambiguous grammars by parser generators for LR grammars), syntax-directed translation, symbol tables, run time storage administration, and error detection and recovery during parsing.

The book also contains a comprehensive three-chapter treatment of code optimisation. The usefulness of the extensive bibliography is enhanced by the notes and references at the end of each chapter. These give valuable references to further works of relevance to the topics covered in the chapter.

Unfortunately the book's suitability as a text for an undergraduate course in compiling is somewhat reduced by the lack of a detailed case study of a working compiler. A companion volume containing the authors' own implementation of the project which they suggest in Appendix B—the building of a compiler for a subset of PASCAL —would remedy this. Another important omission is any treatment of the relationship between compilers and debugging aids. Finally, the book is marred by several typographical errors which could be misleading to the uninformed reader.

Despite these shortcomings the book will, as the authors claim, be useful as a source of ideas and techniques for software designers working both within and outside the field of compiler design.

BERNARD SUFRIN (Oxford)