SELF-ASSESSMENT PROCEDURE XXI

A self-assessment procedure on concurrency

by Brian A. Rudolph

What is Self-Assessment Procedure XXI?
This is the 21st self-assessment procedure. All the previous ones are listed on the facing page. The first 13 are available from ACM* in a single loose-leaf binder to which later procedures may be added.

This procedure is intended to allow computer professionals to test their knowledge of the general concepts of concurrency, that is, the parallel execution of several processes or tasks. It includes questions concerning terminology, formal topics, specifying concurrency, process coordination, and classical problems. In all except a few cases, there is supposed to be only one correct answer for each of the multiple-choice questions.

The next few paragraphs repeat the introduction and instructions given with earlier procedures. Those who read them before may advance directly to the questions.

What is Self-Assessment?
Self-assessment is based on the idea that a procedure can be devised that will help a person appraise and develop his or her knowledge about a particular topic. It is intended to be an educational experience for a participant. The questions are only the *beginning* of the procedure. They are developed to help the participant think about the concepts and decide whether to pursue the matter further.

* Order number 203840. ACM Order Dept., P.O. Box 64145, Baltimore, MD 21264. Members—$15. Non-members—$25.

The primary motivation of self-assessment is *not* for an individual to satisfy *others* about his or her knowledge; rather it is for a participant to appraise and develop his or her own knowledge. This means that there are several ways to use a self-assessment procedure. Some people will start with the questions. Others will read the answers and refer to the references first. These approaches and others devised by the participants are all acceptable if at the end of the procedure the participant can say, "Yes, this has been a worthwhile experience" or "I have learned something."

How to Use the Self-Assessment Procedure
We suggest the following way of using the procedure, but as noted earlier, there are others. This is not a timed exercise; therefore, plan to work with the procedure when you have an hour to spare, or you will be shortchanging yourself on this educational experience. Go through the questions, and mark the responses you think are most appropriate. Compare your responses with those suggested by the Committee. In those cases where you differ with the Committee, look up the references if the subject seems pertinent to you. In those cases in which you agree with the Committee, but feel uncomfortable with the subject matter, and the subject is significant to you, look up the references.

Some ACM chapters may want to devote a session to discussing this self-assessment procedure or the concepts involved.

The Committee hopes some participants will send comments.

Previous Self-Assessment Procedures

Self-Assessment Procedure I
Three concept categories within the programming skills and techniques area
May 1976

Self-Assessment Procedure II
System organization and control with information representation, handling, and manipulation
May 1977

Self-Assessment Procedure III
Internal sorting
September 1977

Self-Assessment Procedure IV
Program development tools and methods, data integrity, and file organization and processing
February 1978

Self-Assessment Procedure V
Database systems
Peter Scheuermann and C. Robert Carlson
August 1978

Self-Assessment Procedure VI
Queueing network models of computer systems
J. W. Wong and G. Scott Graham
August 1979

Self-Assessment Procedure VII
Software science
M. H. Halstead and Victor Schneider
August 1980

Self-Assessment Procedure VIII
The programming language Ada
Peter Wegner
October 1981

Self-Assessment Procedure IX
Ethics in computing
Edited by Eric A. Weiss, from a book by Donn B. Parker
March 1982

Self-Assessment Procedure X
Software project management
Roger S. Gourd
December 1982

Self-Assessment Procedure XI
One part of early computing history
Eric A. Weiss
July 1983

Self-Assessment Procedure XII
Computer architecture
Robert I. Winner and Edward M. Carter
January 1984

Self-Assessment Procedure XIII
Binary search trees and B-Trees
Gopal K. Gupta
May 1984

Self-Assessment Procedure XIV
Legal issues of computing
Jane P. Devlin, William A. Lowell, and Anne E. Alger
May 1985

Self-Assessment Procedure XV
File processing
Martin K. Solomon and Riva Wenig Bickel
August 1986

Self-Assessment Procedure XVI
Computer organization and logic design
Glen G. Langdon, Jr.
November 1986

Self-Assessment Procedure XVII
ACM
Eric A. Weiss
October 1987

Self-Assessment Procedure XVIII
Data Communications
John C. Munson
March 1988

Self-Assessment Procedure XIX
Copyright Law
Riva W. Bickel
April 1989

Self-Assessment Procedure XX
Operating Systems
J. Rosenberg, A. L. Ananda, and B. Srinivasan
February 1990

Self-Assessment Procedure XXI

Contents

Part I. Questions

GENERAL CONCEPTS

The notion of a *process* (often termed a *task*) is difficult to capture precisely, while at the same time being of foremost importance in the study of concurrency. The first part of this self-assessment procedure is about fundamental processes and concurrency concepts.

1. Processes may be described as *physically concurrent* or *logically concurrent*. The distinction between these types of concurrency is analogous to the distinction between:

a. Synchronous processes and asynchronous processes.
b. Real processors and virtual processors.
c. Explicit parallelism and implicit parallelism.
d. Batch processing and interactive processing.

2. Physical concurrency in a uniprocessor system:

a. Can occur among operating system routines.
b. Can occur with peripherals that communicate via interrupts.
c. Occurs through context switching.
d. Cannot occur.

3. Within a process, a *critical section* consists of any sequence of instructions that:

a. Produce a result necessary for the successful completion of a subsequent instruction.
b. Are frequently executed on behalf of the process.
c. Must be accessed on a mutually exclusive basis.
d. Have a high-priority value associated with them.

4. When the result of a computation depends on the speed of the processes involved, there is said to be:

a. Process syncopation.
b. A time lock.
c. Cycle stealing.
d. A race condition.

5. The relative speed of a process in execution cannot reliably be determined due primarily to the variations in:

a. The timing of context switching among processes.
b. The speed of system hardware.
c. The number of instructions included in a process.
d. The types of the instructions included in a process.

6. The major distinction between *lightweight* and *heavyweight* processes centers around:

a. The amount of memory that must be allocated to the process.
b. The average number of instructions executed by the process.
c. The amount of overhead associated with process creation and context switching.
d. The number of I/O requests made by the process.

7. A technique which derives parallelism by decomposing a task into a number of distinct stages which may be overlapped in an assembly-line fashion is called:

a. Phase transition.
b. Pipelining.
c. Linear displacement.
d. Fragmentation.

FORMAL CONCURRENCY TOPICS

The study of concurrency is based on formal models of concurrent computation and the properties of their parallel algorithms. Many attempts have been made to introduce formalisms that are both sufficiently powerful and clear, but none have become dominant. This section focuses on a sampling of these models and on other formal aspects of concurrency.

8. A sequential algorithm with input size i performs

$W(i)$ operations in the worst case. Given a machine with n identical processors, the *best* worst case complexity of a parallel implementation of this algorithm would be on the order of:

a. $W(i)/\log n$.
b. $W(i)/\sqrt{n}$
c. $W(i)/n$
d. $W(i)/n^2$

9. Bernstein's conditions use the concepts of *read sets* and *write sets* to determine:

a. If a group of processes and resources have become involved in a deadlock.
b. If there are any processes suffering from indefinite postponement.
c. If a resource allocation request can be granted safely.
d. If performing multiple operations in parallel will preserve determinacy.

10. The NC problem class consists of problems that can be solved by a parallel algorithm with polynomially many processors in time proportional to a fixed power of the log of their input size. Which of the following problems is most likely not in class NC?

a. Sorting a list of records on a particular key.
b. Searching an unordered list for the record with the largest key.
c. Weighted average computation.
d. Finding the greatest common divisor of two integers.

11. Flynn's taxonomy of parallel computational models uses the concepts of instruction streams and data streams to determine classification. The conventional view is that no existing computers can be classified as:

a. Single-instruction stream, single data stream (SISD).
b. Single-instruction stream, multiple data stream (SIMD).
c. Multiple-instruction stream, single data stream (MISD).
d. Multiple-instruction stream, multiple data stream (MIMD).

12. The number of messages buffered in Hoare's Communicating Sequential Processes is:

a. Zero.
b. One.
c. Bounded by some $n \geq 1$.
d. Unbounded.

13. Which of the following does not completely distribute through the nondeterministic choice operator under the laws established in Hoare's text on communicating sequential processes?

a. Recursion.
b. Interleaving.
c. Prefixing.
d. Concealment.

14. Petri nets provide a method to mathematically analyze systems that contain concurrent activities. Consider the two Petri net graphs in Figure 1. For an initial node marking of n tokens, which of the following observations regarding the execution of these two Petri nets is incorrect?

a. All places in both Petri nets are *n-safe*.
b. All places in both Petri nets will have been marked during execution.
c. One of the two Petri nets is not *strictly conservative*.
d. Exactly n tokens will remain in each Petri net when execution halts.

15. When message transmission systems send empty messages asynchronously, they are effectively equivalent to:

a. Petri nets.
b. Semaphore-based systems.
c. Finite state machines.
d. Turing machines.

16. In the actor paradigm, computational agents called *actors* communicate by message passing and carry out their actions concurrently. Each actor has a mail address and an associated behavior. Which of the following does not characterize the behavior of an actor?

a. An actor may process multiple communications simultaneously.
b. An actor may process only those tasks whose target corresponds to its mail address.
c. An actor may create new actors upon acceptance of a communication.
d. An actor must compute a replacement behavior upon acceptance of a communication.

17. The *paralation model* is an architecture-independent model for parallel programming that can be combined with any base language to produce a concrete parallel programming language. It consists of a single data structure (called a *field*) and three carefully chosen operators. Which of the following is not an operator in the basic paralation model?

a. Merge.
b. Elementwise evaluation.
c. Move.
d. Match.

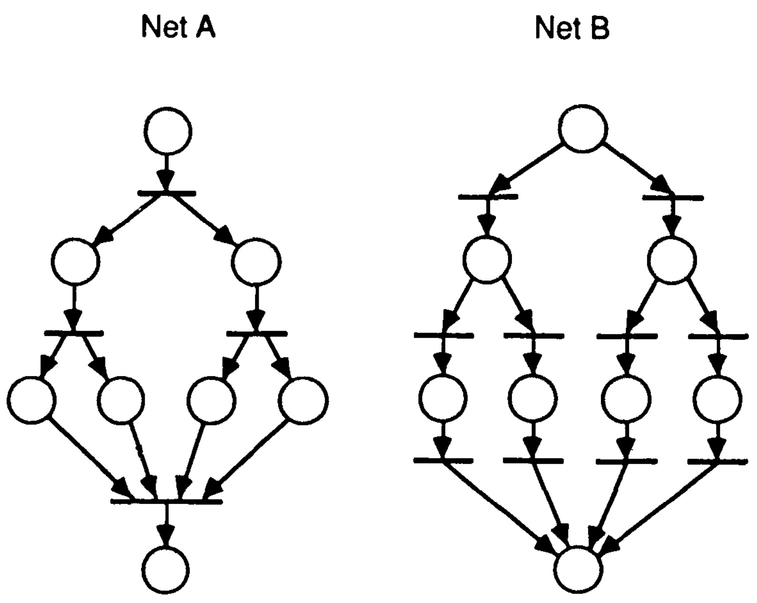

Figure 1. Two Petri-Net Graphs

18. Which model below lends itself most easily to the specification of potentially massive parallelism?

a. Finite state machines.
b. Monitors.
c. Neural networks.
d. Communicating sequential processes.

19. When taking the *axiomatic approach* to the verification of concurrent programs:

a. A concurrent program is transformed to an equivalent sequential program for further analysis.
b. Interleaving scenarios are employed to characterize all possible behaviors of a concurrent program.
c. A functional relationship is established between the initial and final set of values in a concurrent program.
d. Statements in a concurrent program are viewed as relations between predicates in a formal logic system.

SPECIFYING CONCURRENCY
The following questions are concerned with a few of the problems associated with specifying concurrency. The exercises give an opportunity to specify concurrency using some basic primitives.

20. *Coroutines* are not suitable for specifying true parallel processing since:

a. They do not permit the simultaneous execution of multiple processes.
b. They do not provide a well-defined method for transfer of control.
c. They are not sufficiently powerful to implement multiprogramming.
d. They do not provide a mechanism for process synchronization.

21. Statement sequences that cannot be executed in parallel are said to contain *dependences.* Consider the program fragment given in Listing 1. Identify the *flow dependences, antidependences,* and *output dependences* present between the statements in this fragment.

Listing 1

$\vdots$

S_1: $W = X + Y$
S_2: $X = W - Z$
S_3: $Y = 3 + W$
S_4: $W = X + Z$

$\vdots$

22. Parallelism inherent in loop structures but not explicitly specified by the programmer can be extracted by a compiler capable of detecting parallelism automatically. In the absence of dependences, such a compiler can restructure the loop and ultimately create a vector statement for each individual case. This type of transformation is called:

a. Loop optimization.
b. Loop distribution.
c. Loop blocking.
d. Loop interchange.

23. Express the precedence graph in Figure 2 as a concurrent program using **fork**, **join**, and **quit** primitives. The program should permit maximum parallelism.

24. Express the precedence graph in Figure 2 as a concurrent program using the **parbegin/parend** (also known as **cobegin/coend**) concurrent statement. The program should again permit maximum parallelism.

25. Suppose an edge from node *S3* to node *S4* were added to the precedence graph in Figure 2. What effect would this have on the programs developed in questions 23 and 24?

a. The modification would have no effect on either program.
b. The **fork**, **join**, and **quit** primitives could not be used to derive maximum parallelism.
c. The **parbegin/parend** concurrent statement could not be used to derive maximum parallelism.
d. Neither construct could be used to derive maximum parallelism.

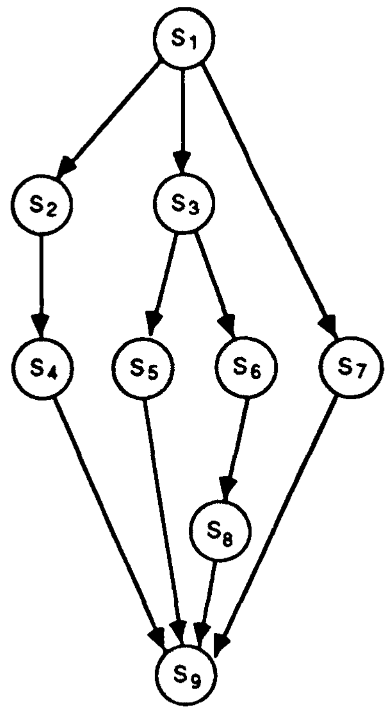

Figure 2. A Precedence Graph

PROCESS COORDINATION

Processes must often synchronize and communicate to accomplish their tasks. Process coordination problems provide one of the most intellectually challenging aspects of concurrency. This portion of the self-assessment procedure examines some problems and concerns associated with these issues.

26. Which of the following is not a proper statement concerning *critical regions*?

a. Critical regions involving distinct data may be executed concurrently.
b. One critical region cannot be nested inside another critical region.
c. A process should remain inside a critical region for only a finite amount of time.
d. A process should not be permitted to terminate inside a critical region.

27. *Starvation* occurs when:

a. At least one process is continually passed over and not permitted to execute.
b. The priority of a process is adjusted based upon its length of time in the system.
c. At least one process is waiting for an event that will never occur.
d. Two or more processes are forced to access critical data in strict alternation with each other.

28. A common assumption underlying mutual exclusion algorithms in shared memory systems is that:

a. Time-critical threats of process starvation can effectively be ignored.
b. A memory reference to an individual word is mutually exclusive.
c. A single instruction executes faster than a group of instructions.
d. A process executing a busy wait will receive a lower scheduling priority.

29. An objection to Dekker's two-process mutual exclusion algorithm concerns the fact that:

a. It relies on race conditions to achieve mutual exclusion.
b. It does not prevent the indefinite postponement of a process.
c. It cannot be generalized to provide mutual exclusion among more than two processes.
d. It uses a common variable that can be altered by any process.

30. Consider busy waiting for entry into a critical section in a shared memory system. In which scenario (or scenarios) below would this not be considered an unreasonable approach?

a. When there are few CPU-bound processes in existence.
b. When a dedicated processor can be assigned to perform the busy wait.
c. When the expected wait is less than the time needed to perform a context switch.
d. None of the above—busy waiting is always unreasonable since it does nothing but waste processor cycles.

31. The first mutual exclusion algorithm independent of any centralized device serialized the requests from competing processes desiring entry into a critical section. This famous algorithm is known as:

a. Conway's Algorithm.
b. Schott's Algorithm.
c. Lamport's Bakery Algorithm.
d. Dijkstra's Banker's Algorithm.

32. In shared memory systems, process synchronization can be supported by special hardware instructions that perform multiple actions atomically such as the reading and modification of a single memory location. Many of these instructions are termed *blocking* since they can be executed only by one process at a time. Conversely, *nonblocking* primitives permit many processors to access a shared variable simultaneously and obtain unique results. Which of the following hardware primitives, when used in conjunction with an interconnection network that can combine requests bound for the same memory location, is nonblocking?

a. The *test-and-set* instruction.
b. The *fetch-and-add* instruction.
c. The *lock* instruction.
d. The *swap* instruction.

33. A *general* (or *counting*) semaphore:

a. Provides less synchronization capability than a binary semaphore.
b. Provides synchronization capability equivalent to a binary semaphore.
c. Provides more synchronization capability than a binary semaphore.
d. Bears no comparable relationship to a binary semaphore.

34. A *condition* in a monitor is associated with:

a. An integer variable, which is initially zero.
b. A binary semaphore, which is initially zero.
c. A queue, which is initially empty.
d. A boolean variable, which is initially false.

35. A monitor has all of the following advantages over a semaphore except:

a. Being a more powerful synchronization construct than a semaphore.
b. Being a higher-level synchronization construct than a semaphore.
c. Providing automatic mutual exclusion within its boundaries.
d. Collecting all routines that modify a set of shared data into one location.

36. One advantage of *path expressions* over monitors is that path expressions:

a. Are more easily implemented within a compiler.
b. Can be used to convey condition synchronization.
c. Allow the specification of an ordering in which to resume blocked processes.
d. Eliminate the need for explicit synchronization code.

37. When working with message-passing systems, *time-outs* are used to:

a. Limit the number of times that a message may be transmitted.
b. Determine that a transmitted message has become lost.
c. Temporarily suspend the transmission of messages.
d. Limit the size of a transmitted message.

38. A distinct advantage of message-passing over semaphores is that:

a. Message-passing is readily extensible to a distributed environment.
b. Message-passing primitives do not necessarily block a process when executed.
c. Message-passing imposes a hierarchical structure on the design of an operating system.
d. Processes engaged in message-passing cannot become involved in a deadlock.

39. In many cases, a group of cooperating processes must all arrive at a common location before any of them are permitted to proceed. This location is termed:

a. An artificial rendezvous point.
b. A barrier synchronization point.
c. A conditional critical region.
d. A synchronization bottleneck.

40. *Event counts* and *sequencers* can be used to solve the bounded-buffer producer/consumer problem:

a. Without requiring mutual exclusion between the producer and consumer processes.
b. Only in the case of a single producer process and a single consumer process.
c. Only when the producer and consumer

processes run at the same relative speed.
d. Only when the information transmitted between the producer and consumer processes is passed by value.

41. A major distinction between *tightly-coupled* systems and *loosely-coupled* systems is that process synchronization:

a. Is simplified in a loosely-coupled system since all processors in the system can access a shared global memory.
b. Is simplified in a loosely-coupled system since a single operating system must control all processors in the system.
c. Is more difficult in a loosely-coupled system since there is only minimal message traffic between processors.
d. Is more difficult in a loosely-coupled system since there is typically no shared memory or clocks.

42. Transaction processing systems such as airline reservation systems must provide a mechanism, which guarantees that each transaction is immune from interference by other transactions that may be occurring at the same time. *Two-phase* transactions obey a protocol that insures this atomicity. In two-phase transactions:

a. All read operations occur before the first write operation.
b. All lock actions occur before the first unlock action.
c. A shared lock on an object must be obtained before an exclusive lock on the object can be obtained.
b. Any currently locked object must be unlocked before another object can be locked.

43. Remote procedure calls can be specified in one of two ways: either as a procedure declaration that is implemented as a server process or as a special statement. The Ada *rendezvous* takes the latter approach by requiring the server side to execute:

a. A **call** statement.
b. A **null** statement.
c. An **accept** statement.
d. An **entry** statement.

44. Which of the following is not an advantage of the Ada rendezvous mechanism?

a. One server can provide multiple services.
b. Communication between the client and server is guaranteed within a finite amount of time.
c. A server can achieve different effects for client calls to the same service.
d. Client calls can be serviced at times determined by the server.

45. The Ada **select** statement is an example of a *guarded command*. The guards:

a. Prevent conflicting tasks from simultaneously executing the **select** statement.
b. Provide mutual exclusion among the alternatives in the **select** statement.
c. Determine which **select** alternatives can be executed.
d. Determine which specific event will be serviced.

46. Which of the following is not a characteristic of an algorithm that exhibits *large-grain parallelism*?

a. It performs relatively many operations between synchronizations.
b. It generates an abundance of message traffic.
c. It can typically take advantage of additional processors as the size of a problem increases.
d. It can be implemented on both multiprocessor systems and multicomputer systems.

CLASSIC CONCURRENCY PROBLEMS

This segment is a collection of problems and exercises, which have become classics in concurrency.

47. Dijkstra's Dining Philosophers problem involves a group of five philosophers whose existence is based solely on two activities: thinking and eating. The philosophers sit around a circular table. In the center of the table is a bowl of spaghetti, which is constantly replenished (the bowl is never empty). The only eating utensils available are five forks. One of the forks is located between each adjacent pair of philosophers. A hungry philosopher, therefore, must acquire the forks to the immediate left and right in order to eat. The life of a philosopher constantly cycles between the thinking and eating states. Consider designing a concurrent program that simulates the activities of the philosophers without severely inhibiting their actions. The primary task in developing an acceptable solution to this problem concerns:

a. Selecting an appropriate mutual exclusion primitive.
b. Avoiding deadlock and process starvation.
c. Selecting an appropriate representation for the resources.
d. Serializing the use of the resources.

48. The Cigarette Smokers' problem consists of an *agent* process and three *smoker* processes. The agent has access to an infinite supply of the three commodities necessary to make and use a cigarette: paper, tobacco, and matches. One of the smoker processes has access to an infinite supply of paper, another has access to an infinite supply of tobacco, and the third has access to an infinite supply of matches. The agent begins by placing two of the three commodities on the table. The smoker process with the missing ingredient must then acquire the two commodities on the table, make and smoke a cigarette, and notify the agent when it has completed. The process then repeats with the agent placing two more of its commodities on the table. This is an example of a synchronization problem that cannot be solved:

a. Using a Petri net.
b. Using only ordinary semaphore operations.
c. Using only asynchronous message passing.
d. Using any model of parallel computation.

Questions 49 and 50 refer to Listing 2 in which a producer and a consumer process synchronize to share a common buffer.

Listing 2

```
var
    mutex, item_available: semaphore;

procedure producer;

begin
  repeat
    produce_item;
    P(mutex);
    append_to_buffer;
    V(mutex);
    V(item_available)
  until false
end;

procedure consumer;
begin
  repeat
    P(item_available);
    P(mutex);
    retrieve_from_buffer;
    V(mutex);
    consume_item
  until false
end;

begin
  semaphore_init(mutex, 1);
  semaphore_init(item_available, 0);

  parbegin
    producer;
    consumer
  parend
end.
```

49. What would be the effect of executing the program in Listing 2 with the two *P* operations in the consumer process interchanged?

a. The producer and consumer processes could deadlock.
b. The producer or consumer process could become a victim of starvation.
c. Mutual exclusion would be violated.
d. The modification would have no effect on the correctness of the program.

50. What would be the effect of executing the program in Listing 2 with the two *V* operations in the producer process interchanged?

a. The producer and consumer processes could deadlock.
b. The producer or consumer process could become a victim of starvation.
c. Mutual exclusion would be violated.
d. The modification would have no effect on the correctness of the program.

51. Determine the proper lower-bound and upper-bound on the final value of the shared variable *tally* output by the concurrent program in Listing 3. Assume that the processes can execute at any speed and that a value can only be incremented after it has been loaded into an accumulator by a separate machine instruction.

Listing 3

```
const
  n = 50;

var
  tally: integer;

procedure total;

var
  count: integer;

begin
  for count := 1 to n do
      tally := tally + 1
end;

begin
  tally := 0;

  parbegin
    total;
    total
  parend;

  writeln(tally)
end.
```

Suppose that an arbitrary number of these increment processes are permitted to execute in parallel under the previous assumptions. What effect will this modification have on the range of final values of *tally*?

52. When the concurrent processes in Listing 4 are executing, what relationship will exist between the values of the shared variables count1 and count2? Again assume that the processes can execute at any speed and that a value can only be incremented after it has been loaded into an accumulator by a separate machine instruction.

a. `count1` and `count2` will remain equal.
b. `count1` will remain greater than or equal to `count2`.
c. `count2` will remain greater than or equal to `count1`.
d. The values of `count1` and `count2` will exhibit no consistent relationship.

Listing 4

```
var
  blocked: array [0..1] of boolean;
  turn, count1, count2: integer;

procedure process(id: integer);
begin
  repeat
    blocked[id] := true;
    while turn <> id do
      begin
        while blocked[1 - id] do;
        turn := id
      end;
    count1 := count1 + 1;
    blocked[id] := false;
    count2 := count2 + 1
  until false
end;

begin
  blocked[0] := false; blocked[1] := false;
  turn := 0; count1 := 0; count2 := 0;

  parbegin
    process(0);
    process(1)
  parend

end.
```

Part II. Suggested Responses

GENERAL CONCEPTS

1. b [6, pp. 36–37; 32, pp. 62–65; 19, p. 22]
2. b [32, pp. 12–15; 6, pp. 296–300; 9, pp. 26–27]
3. c [6, pp. 46–48; 32, pp. 83–84; 9, pp. 76–77]
4. d [19, p. 19; 2, p. 366]
5. a [5, pp. 22–24; 34, pp. 70–72; 19, p. 22]
6. c [2, pp. 272–273; 13, p. 338]
7. b [28, pp. 9–10, 13–15; 9, pp. 29–30; 15, p. 270]

FORMAL CONCURRENCY TOPICS

8. c [28, pp. 42–44, 131; 4, p. 363; 15, p. 260]
9. d [21, pp. 12–14; 26, pp. 52–53, 70–73]
10. d [8, pp. 2–22; 4, pp. 365–366; 15, pp. 271–275]
11. c [2, pp. 19, 111–112; 28, pp. 16–17; 9, pp. 317–318] Pipelined vector processors are sometimes classified as MISD, however.
12. a [17, pp. 134–135, 142; 18, p. 285; 32, pp. 125–127, 132–133]
13. a [17, pp. 101–104, 111–112, 119–120] The recursion operator is not distributive through nondeterministic choice except in the trivial case where identical operands are supplied.
14. b [26, pp. 16–21, 40, 81–84] Since the first three transitions in Net A cause a mark to visit every place by creating an additional token after each firing, Net A is not *strictly conservative*. The final transition removes these duplicate tokens, leaving n tokens in the final place when execution halts. The arrangement of the places and transitions, however, limits the number of tokens in a given place to n, thus making them *n-safe*.

The execution of Net B contains a sequence of transitions that are in *conflict*, causing each token to be preserved as it follows one "path" of transitions from the initial place to the final place. Since no tokens are created or destroyed, Net B is strictly conservative, all places are *n-safe*, and n tokens are left in the final place when execution halts. But the firing of transitions is nondeterministic, so there is no guarantee that all places in Net B will have been marked.

15. b [26, pp. 220–226; 19, pp. 33–36; 7, p. 93]
16. a [1, pp. 23–25]
17. a [31, pp. 7–8, 11–36]
18. c [30, pp. 129–136; 11, pp. 170–187; 14, p. 161]
19. d [23, pp. 319–340; 24, pp. 279–285; 3, pp. 6–9, 32–33, 52–53; 29, pp. 56–58; 10]

SPECIFYING CONCURRENCY

20. a [5, pp. 30–32; 3, p. 10; 7, p. 131]
21. [25, pp. 1184–1187; 27, pp. 15–18; 22, pp. 2-10–2-12]

Flow Dependences:

W:	$S_1\delta S_2$, $S_1\delta S_3$
X:	$S_2\delta S_4$

Antidependences:

W:	$S_2\bar{\delta}S_4$, $S_3\bar{\delta}S_4$
X:	$S_1\bar{\delta}S_2$
Y:	$S_1\bar{\delta}S_3$

Output Dependence:

W:	$S_1\delta^{o}S_4$

22. b [27, pp. 21–27; 2, pp. 226–229; 9, pp. 322–323]
23. [6, pp. 42–44; 2, pp. 154–157; 19, pp. 18–19]

Sample Solution:

```
Threads := 4;
S1; fork Process1; fork
```

```
Process2;   fork Process3; quit;
Process1:   S2; S4; join Threads,Continue;
            quit;
Process2:   S3; fork Process4; S5; join
            Threads,Continue; quit;
Process3:   S7; join Threads,Continue;
            quit
Process4:   S6; S8; join Threads,Continue;
            quit;
Continue:   S9; quit
```

24. [2, pp. 154–158; 7, pp. 57–60; 6, pp. 40–42]

Sample Solution:

```
S1;
parbegin
  begin
    S2; S4
  end;

  begin
    S3;
    parbegin
      S5;
      begin
        S6; S8
      end
    parend
  end;

  S7
parend;
S9
```

25. c [6, pp. 40–41; 2, pp. 54, 57; 3, pp. 11–12]

PROCESS COORDINATION

26. b [32, pp. 107–110; 7, pp. 83–86; 13, pp. 291–292]
27. a [32, pp. 99–100; 13, p. 275; 15, pp. 275–283]
28. b [5, p. 8; 29, p. 40; 13, p. 276; 7, p. 87]
29. d [9, pp. 84–85; 5, pp. 38–39, 43; 29, pp. 18–22]
30. b, c [13, pp. 283–284; 6, pp. 56–57; 2, pp. 162–163]
31. c [29, pp. 50–52; 5, pp. 44–46, 93; 21, pp. 207–208; 9, p. 87] Each process "takes a ticket" (chooses a number in a non-decreasing fashion) when desiring to enter a critical section, similar to the numbered ticket system employed in a busy bakery.
32. b [32, pp. 92–95; 29, pp. 40–44; 2, pp. 162–167]
33. b [16, pp. 7–8; 6, pp. 56–57; 19, pp. 30–31]
34. c [5, pp. 73–78; 6, pp. 62–66; 19, pp. 98–99]
35. a [5, pp. 86–88; 32, pp. 120–121; 6, pp. 65–66]
36. d [6, pp. 68–72; 13, pp. 306–308; 3, pp. 36–39]
37. b [32, pp. 135–136; 21, p. 189; 2, p.169]
38. a [7, pp. 127–130; 6, pp. 72–73; 5, p. 93; 34, p. 105]
39. b [2, pp. 165–166; 22, pp. 2–16; 27, pp. 44–45, 86]
40. a [13, pp. 302–305; 21, pp. 17–23]
41. d [28, pp. 35–42; 32, p. 415; 9, pp. 329–330]
42. b [33, pp. 380–381; 13, pp. 240–241; 21, pp. 244–246]
43. c [3, pp. 48–51, 56–57; 21, pp. 82–92; 6, pp. 78–82; 9, pp. 125–127; 2, pp. 167–169]
44. b [3, pp. 49–50; 5, pp. 93–105; 28, pp. 70–71]
45. c [32, pp. 123–125; 5, pp. 99–104; 21, pp. 85–88; 9, pp. 127–129]
46. b [12, pp. 21–22; 28, pp. 60–61]

CLASSIC CONCURRENCY PROBLEMS

47. b [19, pp. 115–121; 13, pp. 136–138; 21, pp. 30–31]
48. b [26, pp. 192–193, 216–217, 224–226; 5, pp. 71–72; 21, pp. 31–33]
49. a [34, pp. 66–70; 5, pp. 58–60]
50. d [34, pp. 66–70; 5, pp. 58–60]
51. [5, pp. 7–8, 17; 7, pp. 82–83; 13, p. 313] On casual inspection it appears that *tally* will fall in the range $50 \leq tally \leq 100$ since from 0 to 50 increments could go unrecorded due to the lack of mutual exclusion. The basic argument contends that by running these two processes concurrently we should not be able to derive a result lower than the result produced by executing just one of these processes sequentially. The logic of it all is quite appealing. But consider the following interleaved sequence of the load, increment, and store operations performed by these two processes when altering the value of the shared variable:

(1) Process **A** loads the value of *tally*, increments *tally*, but then loses the processor (it has incremented its accumulator to 1, but has not yet stored this value).

(2) Process **B** loads the value of *tally* (still zero) and performs forty-nine complete increment operations, losing the processor after it has stored the value 49 into the shared variable *tally*.

(3) Process **A** regains control long enough to perform its first store operation (replacing the previous *tally* value of 49 with 1) but is then immediately forced to relinquish the processor.

(4) Process **B** resumes long enough to load 1 (the current value of *tally*) into its accumulator, but then it too is forced to give up the processor (note that this was process **B**'s final load).

(5) Process **A** is rescheduled, but this time it is not interrupted and runs to completion, performing its remaining 49 load, increment, and store operations which subsequently sets the value of *tally* to 50.

(6) Process **B** is reactivated with only one increment and store operation left to perform before it terminates. Since it has already performed its final load, it increments its accumulator to 2 and stores this value as the final value of the shared variable! Both processes have terminated, but the value of *tally* has fallen considerably short of 50.

Is 2 the absolute lower bound? It would appear so. Process **A** must perform a store operation in order to corrupt the value of the shared variable, which Process **B** has incremented to 49. This means that Process **A** had to perform at least one increment operation, implying that 49 will be replaced by a minimum value of 1. Once Process **B** has loaded this value into its accumulator, it must still perform a final increment before this value can be stored. Thus 2 is the proper lower bound.

All values in the range 2 through 49 are likewise potential results. The obvious interleaved sequences cause Process **B** to initially lose the processor after completing 50-k increment cycles ($1 \leq k \leq 48$) in step (2).

Although these sequences would hardly ever occur, they are nonetheless possible. Thus the proper range of final values is $2 \leq tally \leq 100$.

For the generalized case of p increment processes, the proper range of results would be $2 \leq tally \leq 50p$ since it is possible for all other processes to be initially scheduled and run to completion in step (5) before Process **B** would finally destroy their work by finishing last.

52. d [32, p. 141; 29, p. 25; 20, p. 45] This concurrent program is based upon Hyman's incorrect mutual exclusion algorithm. Since mutual exclusion is not properly enforced (both processes can be in their critical sections simultaneously) and the execution speed of the processes cannot be determined, no consistent relationship can be stated.

Part III. Reference Titles

Suggested References

1. Agha, G.A. *Actors: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, Cambridge, Mass., 1986.

2. Almasi, G.S., and Gottlieb, A. *Highly Parallel Computing.* Benjamin/Cummings Publishing Company, Redwood City, Calif., 1989.

3. Andrews, G.R., and Schneider, F.B. Concepts and notations for concurrent programming. In *Concurrent Programming,* N. Gehani and A.D. McGettrick, Eds. Addison-Wesley, Reading, Mass., 1988, pp. 3–69. (First published in *Comput. Surv. 15,* 1 (Mar. 1983), 3–43.

4. Baase, S. *Computer Algorithms.* 2d ed. Addison-Wesley, Reading, Mass., 1988.

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

6. Bic, L., and Shaw, A.C. *The Logical Design of Operating Systems.* Prentice Hall, Englewood Cliffs, N.J., 1988.

7. Brinch Hansen, P. *Operating System Principles.* Prentice Hall, Englewood Cliffs, N.J., 1973.

8. Cook, S.A. A taxonomy of problems with fast parallel algorithms. *Inf. and Cont. 64* (1985), 2–22.

9. Deitel, H.M. *Operating Systems.* 2d ed. Addison-Wesley, Reading, Mass., 1990.

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

11. Feldman, J.A., Fanty, M.A., Goddard, N.H., and Lynne, K.J. Computing with structured connectionist networks. *Commun. ACM 31,* 2 (Feb. 1988), 170–187.

12. Finkel, R.A. Large-Grain Parallelism—Three case studies. In *The Characteristics of Parallel Algorithms,* L.H. Jamieson, D.B. Gannon, and R.J. Douglass, Eds. The MIT Press, Cambridge, Mass., 1987, pp. 21–63.

13. Finkel, R.A. *An Operating Systems VADE MECUM.* 2d ed. Prentice Hall, Englewood Cliffs, N.J., 1988.

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

15. Harel, D. *Algorithmics.* Addison-Wesley, Reading, Mass., 1987.

16. Hemmendinger, D. Comments on "A correct and unrestrictive implementation of general semaphores." *Oper. Syst. Rev. 23,* 1 (Jan. 1989), 7–8.

17. Hoare, C.A.R. *Communicating Sequential Processes.* Prentice Hall, Englewood Cliffs, N.J., 1985.

18. Hoare, C.A.R. Communicating sequential processes. In *Concurrent Programming,* N. Gehani and A.D. McGettrick, Eds. Addison-Wesley, Reading, Mass., 1988, pp. 278–308. (First published in *Commun. ACM 21,* 8 (Aug. 1978), 666–677.)

19. Holt, R.C. *Concurrent Euclid, The UNIX System, and TUNIS.* Addison-Wesley, Reading, Mass., 1983.

20. Hyman, H. Comments on a problem in concurrent programming control. *Commun. ACM 9,* 1 (Jan. 1966), 45.

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

22. Osterhaug, A., Ed. *Guide to Parallel Programming on Sequent Computer Systems.* 2d ed. Prentice Hall, Englewood Cliffs, N.J., 1989.

23. Owicki, S.S., and Gries, D. An axiomatic proof technique for parallel programs. *Acta Inf. 6* (1976), 319–340.

24. Owicki, S.S., and Gries, D. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM 19,* 5 (May 1976), 279–285.

25. Padua, D.A., and Wolfe, M.J. Advanced compiler optimizations for supercomputers. *Commun. ACM 29,* 12 (Dec. 1986), 1184–1201.

26. Peterson, J.L. *Petri Net Theory and the Modeling of Systems.* Prentice Hall, Englewood Cliffs, N.J.,

1981.
27. Polychronopoulos, C.D. *Parallel Programming and Compilers.* Kluwer Academic Publishers, Boston, Mass., 1988.
28. Quinn, M.J. *Designing Efficient Algorithms for Parallel Computers.* McGraw-Hill Book Company, New York, 1987.
29. Raynal, M. *Algorithms for Mutual Exclusion.* The MIT Press, Cambridge, Mass., 1986.
30. Rumelhart, D.E., and McClelland, J.L., Eds. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* Vol. I, *Foundations.* MIT Press/Bradford Books, Cambridge, Mass., 1986.
31. Sabot, G.W. *The Paralation Model.* The MIT Press, Cambridge, Mass., 1988.
32. Silberschatz, A., and Peterson, J.L. *Operating System Concepts.* Addison-Wesley, Reading, Mass., 1988.
33. Ullman, J.D. *Principles of Database Systems.* 2d ed. Computer Science Press, Inc., Rockville, Md., 1982.
34. Whiddett, D. *Concurrent Programming for Software Engineers.* Ellis Horwood Ltd., Chichester, Great Britain, 1987.

Additional References
35. Akl, S.G. *The Design and Analysis of Parallel Algorithms.* Prentice Hall, Englewood Cliffs, N.J., 1989.
36. Bach, M.J. *The Design of the UNIX Operating System.* Prentice Hall, Englewood Cliffs, N.J., 1986.
37. Berstein, A.J. Program analysis for parallel processing. *IEEE Trans. Elect. Comput. EC-15*, 5 (Oct. 1966), 757–762.
38. Bustard, D., Elder, J., and Welsh, J. *Concurrent Program Structures.* Prentice Hall, Englewood Cliffs, N.J., 1988.
39. Chandy, K.M., and Misra, J. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, Mass., 1988.
40. Cherry, G.W. *Parallel Programming in ANSI Standard Ada.* Reston Press, Reston, Va., 1984.
41. Coffman, E.G., and Denning, P.J. *Operating Systems Theory.* Prentice Hall, Englewood Cliffs, N.J., 1973.
42. Crichlow, J.M. *An Introduction to Distributed and Parallel Computing.* Prentice Hall, Englewood Cliffs, N.J., 1988.
43. Desrochers, G.R. *Principles of Parallel and Multiprocessing.* Intertext Publications, Inc., McGraw-Hill Book Company, New York, 1987.
44. Hoare, C.A.R. Monitors: An operating system structuring concept. In *Concurrent Programming*, N. Gehani and A.D. McGettrick, Eds. Addison-Wesley, Reading, Mass., 1988 pp. 256–277. (First published in *Commun. ACM 17*, 10 (Oct. 1974), 549–557.)
45. Hsieh, C.S. Further comments on implementation of general semaphores. *Oper. Syst. Rev. 23*, 1 (Jan. 1989), 9–10.
46. Jamieson, L.H., Gannon, D.B., and Douglass, R.J., Eds. *The Characteristics of Parallel Algorithms.* The MIT Press, Cambridge, Mass., 1987.
47. Krakowiak, S. *Principles of Operating Systems.* The MIT Press, Cambridge, Mass., 1988.
48. Lister, A.M. *Fundamentals of Operating Systems.* 3d ed. Springer-Verlag, New York, 1984.
49. Lorin, H. *Parallelism in Hardware and Software: Real and Apparent Concurrency.* Prentice Hall, Englewood Cliffs, N.J., 1972.
50. Lusk, E., Overbeek, R., et al. *Portable Programs for Parallel Processors.* Holt, Rinehart and Winston Inc., New York, 1987.
51. Madnick, S.E., and Donovan, J.J. *Operating Systems.* McGraw-Hill, New York, 1974.
52. Milenkovic, M. *Operating Systems Concepts and Design.* McGraw-Hill, New York, 1987.
53. Raynal, M. *Distributed Algorithms and Protocols.* John Wiley & Sons, New York, 1988.
54. Reed, D.P., and Kanodia, R.K. Synchronization with event counts and sequencers. *Commun. ACM 22*, 2 (Feb. 1979), 115–123.
55. Theaker, C.J., and Brookes, G.R. *A Practical Course on Operating Systems.* Springer-Verlag, New York, 1983.
56. Turner, R.W. *Operating Systems Design and Implementations.* Macmillan, New York, 1986.
57. Yuen, C.K. *Essential Concepts of Operating Systems.* Addison-Wesley, Reading, Mass., 1986.

Epilogue

Now that you have reviewed this self-assessment procedure and have compared your responses to those suggested, you should ask yourself whether this has been a successful educational experience. The Committee suggests that you conclude that it has only if you have

—discovered some concepts that you did not previously know about or understand, or
—increased your understanding of those concepts that were relevant to your work or valuable to you.

ACM Self-Assessment Procedures Guide for Prospective Authors

Self-assessment procedures are intended to be fairly short mechanisms to help members of ACM appraise and develop their knowledge of subjects important to them in their roles as computer professionals. The purpose of the procedures is tutorial. The subjects of the procedures should be about computing, of widespread interest or importance to ACM members, and comprehensible to the average ACM member after a reasonable amount of effort. However, the subjects need not be of universal interest within the ACM community. The procedure need not present a balanced view of all known ways of solving or viewing a particular problem as long as the procedure is accurate.

The procedure should be aimed at the general ACM membership, not at specialists. The set of items in the procedure seldom would make a good graduate student examination, although some of the items conceivably might be used in such a context.

It is important to keep in mind that the self-assessment procedure is not intended as a test or certification of knowledge for anyone other than the person reading the procedure.

The items in the procedure should be of widely varying difficulty; a few should be easy enough for virtually any ACM member to answer or make a reasonable guess at. The author should supply about 30 items, some or all of which may be based on short examples placed in the procedure. Most of the items in published procedures have been in multiple-choice form, but this is not necessary as long as reasonably short responses can be provided. Some items have had more than one correct response, which is fine as long as the item is appropriately worded. It is suggested that the items not be arranged in order of increasing difficulty and that some easy items appear very near or at the beginning, and occasionally throughout.

Responses should be provided for almost all of the items. Occasionally, an open question might be included (a procedure consisting entirely of open questions would be unusual).

Every item and its response should be associated with a reference. These references should be as precise as possible (including page and, if appropriate, line or paragraph number). References should be only to a few publicly available documents. One should be able to obtain the references without having access to a huge library. If the author can find no references for a response, this probably indicates that the subject or item is too new to appear in a self-assessment procedure.

It is desirable to provide an additional short bibliography for readers who become interested enough to read further. If a good bibliography has already been published, a reference to it should be included as well.

Authors of published procedures have found it useful to test the procedures by asking colleagues and students to work them through. The Committee strongly recommends that this be done prior to submission of a draft.

Please supply the ACM Self-Assessment Committee with your proposed procedure including the following sections: items, responses, references for each item, and bibliography. The Committee will review your procedure and will get technical reviews by experts as needed. If the Committee accepts your procedure, it may ask you to attend a committee meeting to go over any proposed changes. After the authors of accepted procedures sign copyright agreements, the Committee will have the procedure published with an appropriate introduction in *Communications*. The authors of the procedure will be listed as such, as with other *Communications* articles. The membership of the Committee will be listed as part of the procedure.

Author's Address: Brian A. Rudolph, Dept. of Computer Science, University of Wisconsin–Platteville, 421 Pioneer Tower, Platteville, WI 53818.

CONTACT: Neal S. Coulter
Department of Computer Science
Florida Atlantic University
Boca Raton, FL 33431