Subject: Are critical sections good enough?
From: Van Snyder

1 Introduction

If one wants to provide that one image has exclusive access to a variable, no matter from where it is referenced, one cannot simply use a CRITICAL construct, because it is the textual critical construct that limits execution to a single image. One might be tempted to put the critical construct into a procedure, and use the procedure to access the shared variable therein, but the construction of VALUE dummy arguments, and elaboration of the specification part, are not within the critical section.

2 Proposals

2.1 MONITOR procedures

Were it not for VALUE arguments and the generality of specification parts, a MONITOR procedure could be constructed, at the cost of some verbosity, by wrapping the *execution-part* in a CRITICAL construct. But the construction of VALUE dummy arguments, and elaboration of the specification part, cannot be within the critical section.

Monitor procedures should be allowed to be executed from within DO CONCURRENT loops.

2.2 LOCK construct

A CRITICAL construct is a lighter weight exclusion mechanism than a MONITOR procedure, but critical sections provide exclusion by their textual position. In addition to a MONITOR procedure, it would be desirable to have a lightweight mechanism that can exclude access based upon a binary semaphore, no matter where (textually) the exclusion is requested.

3 Edits

Edits refer to 06-007. Page and line numbers are displayed in the margin. Absent other instructions, a page and line number or line number range implies all of the indicated text is to be replaced by associated text, while a page and line number followed by + (-) indicates that associated text is to be inserted after (before) the indicated line. Remarks are noted in the margin, or appear between [and] in the text.

3.1 Laying the groundwork concerning the DO CONCURRENT construct

---

(3b) Exection of a DO CONCURRENT construct divides the execution sequence into a number of execution sequences that does not exceed the iteration count of the construct. Each such execution sequence proceeds independently through the block of one or more different iterations of the construct until every iteration of the construct has been executed exactly once, at which instant they are recombined into a single execution sequence. 15:22+

When a DO CONCURRENT statement is executed, a separate instance of the *block* of the DO CONCURRENT construct is created for each iteration, and the execution sequence that executes the DO CONCURRENT statement is divided into a number of execution sequences that does not exceed the iteration count. Each instance has an independent set of local unsaved data objects. Each execution sequence independently executes one or more different instances of the block in such a way that each instance is executed once. Each instance ceases to exist when execution of its iteration of the DO CONCURRENT construct completes or execution of the program is terminated. If the program is not terminated, completion of execution of the DO CONCURRENT construct recombines the execution sequences into a single execution sequence. 187:20+ New ¶

[Make the first sentence of the paragraph, the one that begins “The processor shall ensure. . . ”, a separate paragraph, and replace the three instances of “image” in it by “execution sequence”. Within the remainder of the paragraph, replace “image” by “execution sequence”. Within NOTE 8.23 replace the first three instances of “image” in it by “execution sequence”.] 192:15-19+

3.2 MONITOR procedures

[Editor: Replace "PURE" by "PURE or MONITOR procedures".] 189:28

or MONITOR 326:29+

C1246a (R1229) If MONITOR appears, neither ELEMENTAL nor RECURSIVE shall appear. 326:34+

12.8 Monitor procedures 337:13+

A **monitor procedure** is a procedure that does not allow an execution sequence to enter it if one has entered it but not completed execution of it. It is defined by a subprogram for which MONITOR appears in the prefix of the initial subroutine statement or function statement. The execution sequence that is prevented from entering is not terminated; its entry is simply delayed until the execution sequence that is executing the monitor procedure completes execution of it. If several execution sequences simultaneously attempt to enter a monitor procedure, exactly one of them enters it and the others are delayed; which one enters it is processor dependent. If several execution sequences attempt to enter a monitor procedure while another execution sequence is executing it, which one proceeds when the execution sequence that is executing it completes executing it is processor dependent.

3.3 LOCK construct

[Editor: Insert "END LOCK" into the table in alphabetical order.] 30:2+

8.1.3a LOCK construct 180:1-

A **LOCK construct** permits an execution sequence to enter it if its lock variable has a lock status of unlocked, and does not permit the execution sequence to enter if its lock variable has a lock status of locked. When an execution sequence enters a LOCK construct, the lock status of its lock variable becomes locked. When an execution sequence completes execution of a LOCK construct, the lock status of its lock variable becomes unlocked. An execution sequence that is prevented from entering is not terminated; its entry is simply delayed until the execution sequence that is executing the LOCK construct completes execution of it. If several execution sequences simultaneously attempt to enter a LOCK construct, exactly one of them enters it; which one enters it is processor dependent. If several execution sequences attempt to enter a LOCK construct while another execution sequence is executing it, which one proceeds when the execution sequence that is executing it completes executing it is processor dependent.

A LOCK construct completes execution when the END LOCK statement is executed, when control is transferred by a branch within the construct to a branch target outside of the construct, when an EXIT statement that belongs to the construct or one that contains it is executed, or when a CYCLE statement that belongs to a construct that contains the LOCK construct is executed.

[Alternatively, a LOCK construct shall be terminated only by execution of the END LOCK statement or an EXIT statement that belongs to the construct.]

R815a *lock-construct* **is** *lock-stmt*
block
end-lock-stmt

R815b *lock-stmt* **is** [*lock-construct-name* :] LOCK *lock-variable*

R815c *lock-variable* **is** *scalar-variable*

C807a (R815b) The type of the *lock-variable* shall be the derived type SEMAPHORE defined in the ISO_FORTRAN_ENV intrinsic module.

R815d *end-lock-stmt* **is** END LOCK [*lock-construct-name*]

C807b (R815a) If the *lock-stmt* of a *lock-construct* specifies a *lock-construct-name*, the corresponding *end-lock-stmt* shall specify the same *lock-construct-name*. If the *lock-stmt* of a *lock-construct* does not specify a *lock-construct-name*, the corresponding *end-lock-stmt* shall not specify a *lock-construct-name*.

C807c (R815c) The type of the *lock-variable* shall be the derived type SEMAPHORE defined in the ISO_FORTRAN_ENV intrinsic module. The lock variable shall not have the ALLOCATABLE or POINTER attribute, and shall not be a subcomponent of an object that has the ALLOCATABLE or POINTER attribute.]

[Editor: “derived type” ⇒ “derived-type definitions”.] 437:30

13.8.3.5a The SEMAPHORE derived type 439:1-

The type of a *lock-variable* in a LOCK construct (8.1.3a) shall be the SEMAPHORE derived type. The SEMAPHORE derived type has private components, at least one of which has default initialization that indicates that the initial lock status of objects of SEMAPHORE derived type is unlocked.