LITTLE Newsletter #32

P. Shaw

Feburary 4, 1974

Interrupt Handling in LITTLE: Possible Revisions

Contents

1. Introduction

1.1 Overview

This memo outlines a set of possible revisions and additions to the interrupt handling mechanism described in LITTLE Newsletter #30. The basic themes of these changes are:

1) Centralization of scheduling operations;
2) "bundling" of features that are likely to be naturally associated, and "fixing" of features that are unlikely to need to be varied dynamically;
3) introduction of a "test and set" instruction, and use of it and other primitives to program simpler synchronization macro operations; and
4) consideration of checkable facilities (both statically checkable and dynamically checked features are described) for lock enforcement and deadlock prevention.

Some of the resulting language is therefore at a slightly higher semantic level than the rest of LITTLE: several of the proposed operations "package" more function than is common in LITTLE; also, a rudimentary runtime system for simple scheduling and (extremely primitive) dynamic storage allocation is required. The justification offered for these deviations from the LITTLE design philosophy is the difficulty of concurrent programming -- coding and more especially testing. It is felt that the proposed features would simplify the development of interrupt handlers to an extent that would more than make up for the attendant loss of flexibility and increase in implementation overhead.

The remainder of this section provides an overview of the proposed features which are then described in more detail in sections 2 through 5.

1.2 Basic Statements

Section 2 describes the basic set of statements which are proposed. The ENABLE and DISABLE statements are unchanged from #30; PROCESS, ALLOCATE and FREE are substantial revisions of features from #30, which "re-package" the Newsletter #30 functions of PROCESS, SETUP, PSIZE, INITIAL SETUP, PRENABLE/ PREDISABLE, and ATTACH. The BLOCK, WAKEUP, and TSET statements are scheduling and synchronizing primitives not found in #30, which are felt to be important for use in programming the higher level operations of section 3. INT-HANDLER and CURRENT-PS are simple state-query functions which are also required for the operations of section 3.

The #30 operation of RECALL is not explicitly provided, but can be programmed by BLOCK and WAKEUP, as shown in the SIGNAL_INTERRUPT macro (section 3.3). The #30 operation of fetching variable-values from process-states was omitted as hopefully not required.

The remainder of this section provides an informal description of each of the section 2 statements.

1) PROCESS

   This declaration statement, like the PROCESS statement in #30, defines a group of subroutines and namesets as constituting a process. This statement differs from that of #30 in that it includes optional clauses for INITIAL, INT_SOURCE, DISABLED/ ENABLED, and INT_MASK attributes. In #30 these attributes were specified dynamically in the SETUP and ATTACH and PRENABLE/ PREDISABLE statements. It was felt, however, that most processes would be written explicitly for a particular INT_SOURCE, with the requirement that they be entered with a given ENABLE status and INT_MASK. It was therefore decided to make all of those features static attributes of a process.

Note that INT_SOURCE may be either a hardware or software interrupt-source (the codes for INT_SOURCE's are not specified herein). The SIGNAL_INTERRUPT macro (section 3.3) illustrates how interrupts could be raised by a program.

2) ALLOCATE

This is the same as the SETUP statement from #30, with what was felt to be a more accurately suggestive name. It differs from SETUP in that it doesn't have an INITIAL option (that option was moved to the PROCESS declaration), and it doesn't have the initial value clauses -- it was felt that initial values should be taken from those specified in the PROCESS declaration and its constituent subroutine declarations, and that parameter-passing for processes would not be required.

ALLOCATE also differs from the SETUP of #30 in that, if the process has an INT_SOURCE clause, ALLOCATE connects the new process-state to the specified interrupt source. If the interrupt source is already connected to a process-state the existing connection is not broken and an error return is made (the previous connection can be located by an INT_HANDLER statement and broken by a FREE statement).

ALLOCATE also provides feedback in the array-index parameter, indicating either an error return (-1 if the interrupt source was already connected, Ø if the process-state won't fit in the array) or indicating the next available position in the array (or #array+1 if the process-state exactly fills it) -- the latter information eliminates the need for the separate PSIZE statement of #30.

Note that the storage management functions implied are extremely rudimentary. It is assumed that a "loader" will provide storage for all PROCESSes with the INITIAL clause (if this is problematic an allocation-source array clause could be added to the INITIAL PROCESS declaration).

ALLOCATE simply determines if a new process-state will "fit" -- it doesn't maintain the "next position" index or a "free list" and certainly doesn't garbage collect, etc. The "allocation" functions of ALLOCATE (as distinct from the connection of interrupt sources, which ALLOCATE also performs) could be programmed as macros using the PSIZE statement described in #30.

3) FREE

This statement disconnects the specified process state from its interrupt source, then "destroys" it so it cannot be re-used inadvertently.

4) BLOCK

This statement stops execution of the current process. It is assumed that a scheduler then switches to some other process on a "READY" list. The blocked process is not put on that READY list. The macros P and LOCK in section 3 show the use of BLOCK.

5) WAKEUP

This statement adds the specified process-state to the READY list. Macros V and UNLOCK in section 3 show the use of WAKEUP.

6) DISABLE,

7) ENABLE

These are as described in #30. Macros P, V and LOCK, UNLOCK in section 3 show the use of ENABLE and DISABLE.

8) TSET

This is the common test-and-set instruction, which is necessary for coordination of data accesses in multi-processor configurations.

9) INT_HANDLER

This is a query statement which returns a pointer to the process-state, if any, connected to the specified interrupt source. The SIGNAL_INTERRUPT macro in section 3 illustrates its use.

10) CURRENT_PS

This is another query statement, returning a pointer to the currently running process-state. Its use is illustrated in the P and LOCK macros.

1.3 Macro Statements

The basic statements described above provide adequate primitives for synchronizing multiple processes. These primitives are, however, somewhat tricky to use, so that macro statements providing simpler synchronizing capabilities are desirable. Section 3 below describes two sets of such macros -- P and V, and LOCK and UNLOCK. A SIGNAL_INTERRUPT macro is also described. These macros can be summarized as follows:

1) P, V and SEM

The SEM macro generates a data structure for the traditional "semaphore" object. P and V are the usual operations on semaphores, sometimes called DOWN and UP or wait and send.

2) LOCK and UNLOCK

These operations assume that PUBLIC NAMESET's are implemented with an unnamed "lock" field which is accessible to the LOCK and UNLOCK macros. The macros then take as argument a public nameset name and lock or unlock it.
If a LOCK is issued when the public nameset has already been locked by another process, then the process issuing the LOCK is BLOCKed until the other process issues an UNLOCK.

The locking is not enforced, but depends on all users of a public nameset voluntarily issuing LOCK and UNLOCK before and after accessing its elements. Sections 1.4 and 4 discuss lock enforcement.

The locks are made attributes of namesets, rather than an explicit separate data type (in the manner of the SEM macro) in anticipation of the lock enforcement described in sections 1.4 and 4.

The locks are on namesets rather than on individual variables. This is because it is assumed that namesets represent logical groupings of variables such that it is relatively likely one would wish to lock several of the variables at a time, and fairly unlikely that one would need to lock several namesets at a time -- although the latter is not precluded.

No distinction is made between locking for read access versus locking for write access. This is because it is assumed that the public namesets will be "control block" type of data, so that pure read accessing will be rare, and not worth the rather more complicated mechanism of distinct READLOCK's and WRITELOCK's.

3) SIGNAL_INTERRUPT

This macro illustrates the use of the INT_HANDLER and WAKEUP instructions to achieve a simulated interrupt.

1.4 Lock Enforcement

The LOCK and UNLOCK macro statements provide a method of locking public namesets, but do not provide for lock enforcement -- a process could still read or write a public nameset field without having LOCKed the nameset.

It would be advantageous to provide for enforcement of locks. This could be done in two ways: statically or dynamically. Both could probably be implemented with the macro facility (if it has roughly the capability of the PL/I compile-time facility) , but would probably be more easily handled in the basic translator. Sections 4.1 and 4.2 outline respectively a dynamic and a static lock enforcement scheme. Both schemes are of course only as good as the ability of the LITTLE translator to recognize accesses to public nameset data.

1.5 Deadlock Prevention

The LOCK and UNLOCK facilities, whether enforced or not, are completely general, permitting deadlocks to arise if not used carefully. Rudimentary deadlock prevention could easily be added. The simplest approach, quite probably adequate for most "control block" type interactions, would be to permit a process to own at most one lock at a time -- such a restriction could easily be added to either the unenforced mechanism or to the static or dynamic enforcement mechanism.

A slightly more general approach is outlined in section 5. In this approach the programmer defines a partial ordering on the public namesets: those for which deadlock prevention is not to be done are assigned to "level number" zero (the default); those for which deadlock prevention is desired are assigned positive level numbers (in the nameset declaration). The rule is then that a process which currently owns a lock with a positive level number i can only obtain locks which are either level zero or have level numbers strictly greater than i (until the lock at level i is released). Unrelated namesets (those which will never need to be locked simultaneously) can of course be assigned the same level numbers.

Sections 5.1 and 5.2 outline respectively dynamic and static enforcement of that rule.

1.6 Summary

This memo outlines a wide range of facilities, which considerable functional overlap. It is to be hoped that a small selection from among the facilities would be adequate for the intended interrupt handling applications. I would recommend assessment of those applications to determine

1) whether the fixed assocation of PROCESS, INT_SOURCE, DISABLE/ENABLE and INT_MASK is adequate;
2) whether the dynamic creation of process-states (ALLOCATE) is really needed, or whether instead the INITIAL option can be used exclusively;

3) If ALLOCATE is required, is it valid to have it imply connection of the process-state with its INT_SOURCE;
4) Are P, V, and SEM type synchronizations required, or are data locks sufficient;
5) Will the type of shared data be such as to make a one-lock-at-a-time rule workable;
6) If a one-lock rule isn't practiable, are deadlock patterns of use likely;
7) Will there be enough sharing of data to make lock enforcement worthwhile; if so are the restrictions on program structure which are necessary for static checking endurable.

To summarize: asynchronous programs are difficult to understand and worse to test. It therefore seems desirable to introduce structural restrictions which simplify the programs, which reduce the opportunity for error, and which introduce the possibility of automatic validity checks at points of asynchronous interaction.

2. Basic Statement Definitions

2.1 PROCESS declaration

```
[INITIAL] PROCESS processname;
     [INT_SOURCE n;]
     [DISABLED/ENABLED;]
     [INT_MASK b;]

     (nameset declarations)
     (main program)
     (subprocedure-list)

END processname [BEGIN];
```

1) If the INITIAL option is specified then storage for the process-state is allocated implicitly (in unnamed storage) at the time the system is "loaded"; otherwise the process-state storage is allocated by use of an ALLOCATE statement.
2) If the INT_SOURCE option is specified, then the process and process-state are associated with interrupt source n; the association is made when the process-state is (implicitly or explicitly) allocated, and applies to the processor on which the allocation is executed. There can be at most one INITIAL PROCESS having an INT_SOURCE option for any given n.
3) If the DISABLED option is specified then the processor will be set to DISABLED on entry to the process; otherwise it will be ENABLED (i.e. ENABLED is the default). If the INT_MASK option is specified, then on entry to the process the interrupt sources of the processor will be masked as specified by bit-string b; otherwise the interrupt sources will be masked as they were prior to entry to the process.
4) The nameset-declarations, main program and subprocedure list are as described in #30.
5) Exactly one process should have the BEGIN option.

2.2 Allocate Statement

ALLOCATE p IN (x,i)

1) p must be a PROCESS; x an ARRAY; and i an INTEGER
2) If p has an INT_SOURCE option n, then if there is already a process-state associated with interrupt source n on the current processor, then set i to -1 and return.
3) If there is room for a process-state record for p in x beginning at position i, then allocate and initialize one, and set i to either the next location in x or, if this allocation fills x, to #x+1; if there is not sufficient room in x then set i to zero and return.

4) If process p has an INT_SOURCE option n, then associate the new process state with interrupt source n on the current processor.

3. Free Statement

FREE p IN (x,i)

1) DISABLE. If p has an INT_SOURCE option, and if there is a process-state for p at x(i), then dissociate that process-state from the interrupt source on the current processor. ENABLE.
2) If there isn't a process-state record for p at x(i), then set i to -1 and return; otherwise "destroy" the process-state record and return.

2.4 Block Statement

BLOCK

The processor-state of the current processor is stored into the fields of the process-state which is currently running. An entry on the READY list is then selected, and the processor-state is loaded from the corresponding fields of its process-state, thereby starting it to running.

Note that the process issuing BLOCK is not added to the READY list, and will only be "re-activated" by a subsequent WAKEUP issued by some other process.

2.5 Wakeup Statement

WAKEUP ps

ps must be a pointer to a process-state record. A pointer to ps is added to the READY list. It is unspecified whether the scheduler will switch to ps (or to some other process-state on the READY list) or will continue with the current process; if the scheduler were to switch then the current process-state would of course be put on the READY list.

2.6 Disable Statement

DISABLE

The enabling status of the current processor is set to DISABLED. Note that this does not destroy the current interrupt mask.

2.7 Enable Statement

ENABLE

The enabling status of the current processor is set to ENABLED.

2.8 TSET Statement

TSET ℓv, rv

If the value of ℓv is zero, then it is set to 1 and a value of 1 is returned in rv; otherwise a value of Ø is returned in rv.

Note: Execution of this statement is atomic relative to all other processes and processors. It is assumed to be implemented by an identical test-and-set instruction which most systems (certainly those which are available in multi-processor configurations) have, or by a DISABLE-ENABLE expansion on machines which don't have test-and-set.

2.9 Int-handler Statement

INT_HANDLER i, ps

The address of the process-state associated currently with interrupt source i on the current processor is returned in ps; if there is no currently associated process-state then a value of zero is returned in ps.

2.10 Current-ps Statement

CURRENT_PS ps

The address of the current process-state is returned in ps.

3. Macro Statements

3.1 P, V, SEM

1) SEM x(i,n)

This macro generates a data structure as follows:

```
01   X
     02   TS BIN INIT(Ø)
     02   V  BIN INIT(i)
     02   NI BIN INIT(1)
     02   NR BIN INIT(1)
     02   Q(n) BIN
```

X.TS is a lock for the semaphores
X.V is the value of the semaphore, which is initialized to i and which will never be allowed to be less than -n;
Q is an array of addresses of process-state's which are enqueued on the semaphore; NI and NR are indexes of the next positions in Q at which entries should be inserted (NI) or removed (NR).

2) V(S)

S must have been created by the SEM macro.
The expansion is:

```
        DISABLE;
LOOP:   TSET S.TS, TEMP;
        IF TEMP .NE. 1 THEN GOTO LOOP;
        S.V = S.V + 1;
        IF S.V .GE. 1
           THEN DO;  S.TS = Ø;
                     ENABLE;
                     RETURN;
                END;
        WAKEUP S.Q(S.NR);
        S.NR = MOD(S.NR; #S.Q) + 1
        S.TS = Ø;
        ENABLE;
        RETURN;
```

Explanation: fields of a semaphore S should be referenced only by the P and V expansions; the P and V expansions obey the convention that S.TS will be set to 1 if any process is executing a P or V on S, and set to Ø otherwise. The P and V expansions both DISABLE so that a process won't be descheduled while it's in the middle of a P or V. On a uni-processor configuration the TSET will therefore always return a 1 and is thus redundant; on a multiprocessor configuration the TSET will return a zero if a process on another processor is in the middle of a P or V on S -- in this case we have chosen to go into a loop, or "busy wait", since we know that the other processor will set S.TS to zero "soon" -- i.e. in the number of instructions it takes to finish the P or V -- so that the busy wait is probably more efficient than a process-switch.

Other approaches are therefore possible:

i) we could permit processes to be interrupted in the middle of P or V operations;

ii) we could bump the current process and schedule another if the semaphore is busy.

On a uniprocessor (i) implies (ii), since the semaphore which is busy must have been locked by a process which had been interrupted; on a multiprocess configuration the process which is referencing the semaphore in (ii) may be running on another processor, or, only if (i) is permitted, may not be running, so that (i) and (ii) are independent alternatives.

Both (i) and (ii), however, introduce complications which seem certain to offset potential gains in efficiency.

The treatment of DISABLE and TSET in this expansion of the V macro is also followed in the P, LOCK and UNLOCK macros below.

RETURN statements mean exit from the expansion.

3) P(S)

S must have been generated by the SEM macro. The expansion is:

```
        DISABLE;
LOOP:   TSET S.TS, TEMP;
        IF TEMP .NE. 1 THEN GO TO LOOP;
        S.V = S.V - 1;
        IF S.V .GE. Ø
           THEN DO;  S.TS = Ø;
                     ENABLE;
                     RETURN;
                END;
        IF - S.V .GT. # S.Q THEN ERROR;
        CURRENT_PS TEMP;
        S.Q (S.NI) = TEMP;
        S.NI = MOD(S.NI, #S.Q) + 1;
        S.TS = Ø;
        ENABLE;
        BLOCK;
        RETURN;
```

3.2 LOCK, UNLOCK

1) Locks

Public namesets should have an implicit lock associated with them, whose structre would be:

```
01  LOCK
    02   OWNER BINARY INIT(Ø)
    02   TS    BINARY INIT(Ø)
    02   NI    BINARY INIT(1)
    02   NR    BINARY INIT(1)
    02   NQ    BINARY INIT(Ø)
    02   Q(N)  BINARY
```

TS, NI, NR and Q are as in the SEM macro; N is an implementation defined limit; NQ will be the number of entries currently in Q; OWNER will be a pointer to the process state, if any, which currently "owns" the nameset lock.

2) LOCK NS

NS must be a public nameset. Let L be the associated lock. The expansion is:

```
        DISABLE;
LOOP:   TSET L.TS, TEMP;
        IF TEMP .NE. 1 THEN GOTO LOOP;
        CURRENT_PS TEMP;
        IF L.OWNER = 0 OR L.OWNER = TEMP
           THEN DO;  L.OWNER = TEMP;
                     L.TS = Ø;
                     ENABLE;
                     RETURN;
                END;
        IF L.NQ = #L.Q  THEN ERROR;
        L.NQ = L.NQ + 1;
        L.Q (L.NI) = TEMP;
        L.NI = MOD(L.NI, #L.Q) + 1;
```

```
        L.TS = Ø;
        ENABLE;
        BLOCK;
        RETURN;
```

3) UNLOCK NS

NS must be a public nameset. Let L be the associated lock. The expansion is:

```
              DISABLE;
       LOOP:  TSET L.TS, TEMP;
              IF TEMP .NE. 1 THEN GOTO LOOP;
              CURRENT_PS TEMP;
              IF L.OWNER .NE. TEMP THEN ERROR;
              IF L.NQ = Ø
                 THEN DO;  L.OWNER = Ø;
                           L.TS = Ø;
                           ENABLE;
                           RETURN;
                      END;
              L.NQ = L.NQ-1;
              L.OWNER = L.Q (L.NR);
              WAKEUP(L.Q(L.NR));
              L.NR = MOD(L.NR, #L.Q) + 1;
              L.TS = Ø;
              ENABLE;
              RETURN;
```

3.3 Signal-Interrupt Macro

SIG_INT N

Expansion:

```
              INT_HANDLER N, TEMP;
              IF TEMP = Ø THEN ERROR;
              WAKEUP TEMP;
              BLOCK;  /* optional */
```

4. Lock Enforcement

The mechanisms to follow assume that PUBLIC NAMESET's have the implicit lock field described in Section 3.2.

4.1 Dynamic Enforcement

Each time a field in a public nameset is referenced, generate code to dynamically test to see that the current process-state is the OWNER of the implicit lock field of the public nameset.

Optimizations are of course possible: recognizing that the current statement is only reachable along a path that has a LOCK statement. Even without optimization, however, the gain in integrity should justify the increased execution cost of access to public variables.

4.2 Static Enforcement

This involves two new statments:

```
REGION ns;
ENDREGION;
```

The REGION expansion issues a LOCK on the public nameset ns; ENDREGION generates an UNLOCK on the ns specified in the (textually) most recent REGION statement. The REGION and ENDREGION statements must occur paired, in that order, and may be nested (unless it is decided to prohibit multiple locks).

The following restrictions then apply:

i) If a variable v is a field of a public nameset ns, then references to v may only occur between a REGION statement which specifies ns and the corresponding ENDREGION

ii) Branches into and out of a REGION-ENDREGION block are prohibited.

5. Deadlock Prevention

The facilities to follow assume that PUBLIC NAMESET's have the implicit lock field described in Section 3.2. They also assume that the PUBLIC NAMESET declaration is extended to include an optional "level number" integer, perhaps with syntax

```
PUBLIC LEVEL(integer) NAMESET
       name(name-list)
```

If the LEVEL clause is omitted a level number of zero is defaulted.

5.1 Dynamic Checking

This approach assumes either no lock enforcement, or the dynamic enforcement of section 4.1. The approach adds dynamic level number checking logic to the LOCK macro of section 3.2.

Include in each lock item a LEVEL-NO field with the default or explicit level number specified for the nameset, together with a PREVIOUS_LOCK field which will either be zero or contain a pointer to another lock owned by the same process. Include in each process-state record a LAST_LOCK field which will have either zero or a pointer to the lock most recently obtained by the process.

When a LOCK is issued, if the level number in the lock field of the specified nameset is zero then perform no further tests (and don't connect the lock to LASTLOCK); otherwise, check to see that the new level number is strictly greater than the one (if any) pointed to by the LASTLOCK field of the current process-state. Set the PREVIOUS_LOCK field of the new lock to the current value of the LASTLOCK field of the process-state; set the LASTLOCK field to point to the new lock.

On an UNLOCK, first verify that the current process-state is the owner of the specified lock; then unlock all locks owned by the process-state from the one pointed to by LASTLOCK through the one specified in the current UNLOCK; set the LASTLOCK

field of the process-state to the value of the PREVIOUS_LOCK field of the specified lock.

5.2 Static Checking

This approach assumes that the static lock enforcement of section 4.2 is used. The approach does not require the extra fields (LAST LOCK, LEVEL_NO and PREVIOUS_LOCK) which were specified above.

At compile time, maintain a LEVELS integer stack, initially empty. When a REGION statement is processed, if the level number of the specified nameset is zero, then perform no checks; otherwise, ensure that the level number is strictly greater than the number at the top of the LEVELS stack. Then push the new level number onto LEVELS. When an ENDREGION is processed, pop the top entry off the LEVELS stack.