Clock-driven FORTRAN task scheduling in a multiprogramming environment

HOWARD L. KAPLAN

Clinical Research Unit, Addiction Research Foundation of Ontario, 33 Russell Street, Toronto, Ontario, Canada M5S 2S1

The existing ISA-standard FORTRAN task-scheduling calls require careful programming to provide timing accuracy even in a single-user system, and do not explicitly provide a mechanism for resolving task conflicts in a multiprogramming system. A simple assignment of priorities to tasks can guarantee timing accuracy to one task, but the risks generating maximum, unnecessary interference with other tasks. An alternative to such a standard priority scheme, not known to be available on common commercial systems, guarantees timing for one task while minimizing interference among tasks.

Early in 1976, the Addiction Research Foundation of Ontario began investigating real-time computer systems for a proposed new Human Response Laboratory. Our plans included not only on-line control of some rather sophisticated experiments, but also large amounts of data reduction involving waveform analyses. We were interested in a system that would support FORTRAN for both control and analysis tasks. After investigating several medium-scale systems that allow the scheduling of multiple tasks from a shared system timer, we discovered that available system calls were less useful than we preferred in an ideal system.

The particular focus of this discussion is FORTRAN, but many of the same considerations are applicable to ALGOL, to other algebraic languages, and to assembler-coded programs. FORTRAN is merely the most familiar of a class of languages whose surface structure is a strictly sequential execution of tasks, that is, a fetch-execute architecture, rather than a production machine, or recognize-do architecture. This is rather different from a language like PL/I, where there are explicit facilities for sharing control among concurrent tasks within one program. It is also quite different from a state-diagram language in which the system acts upon the first to occur of several alternate events. Whatever features the operating system makes available to FORTRAN are generally available to the other compiled, algebraic languages and to assembler programs, so these remarks are equally applicable to other sequential languages.

THE SINGLE TASK: PROBLEMS AND SOLUTIONS

A logical starting point is one of the existing ISA-standard FORTRAN routines for industrial control,

discussed by Castellan (1976).¹ The most important of these is CALL WAIT(J,K,M), to delay program execution for time J, measured in unit K. To simplify this discussion, we will substitute the notation CALL DELAY(N), to delay program execution for N msec. This DELAY command is similar to that found in a number of other languages; during the delay, the explicit program activity ceases, though previous I/O commands initiated by the program (such as outputting a print buffer or receiving analogue input through a device with its own memory access hardware) may continue. As Creelman (1974) noted, this may provide adequate control if the program never consumes more continuous CPU time than the space between clock ticks, but may lead to inaccuracies if there is a need for heavy CPU activity in setting up the next stimulus. Figures 1 and 2, adapted from Creelman's article, show the essential differences between the "typical" and "preferred" systems; while timing occurs between program events in the typical system, it bridges over CPU activity in the preferred system. In the preferred system, a task can start a programmed delay and then return control to itself to perform I/O or calculations while the clock continues to run. It then schedules the next event to occur at a known time from the last clock reset, rather than at a known time from the variable endpoint of the calculations.

We can now investigate the use of DELAY and alternative routines in a typical benchmark task, using additional subroutines LITON, LITOFF, and SELECT. (All the FORTRAN calls discussed here can be found in Table 1.) Subroutines LITON(L) and LITOFF(L) simply turn light L on or off. Subroutine SELECT(L) is a user-written task taking a variable amount of CPU time between 100 and 200 msec; this task selects one of several alternate stimuli for the next presentation and returns the selection in L. Our basic benchmark task will be to select a light L, turn it on for 500 msec, turn it off for 500 msec, and, while it is off, select the next light to display, repeating this loop 50 times.

Paper presented at the Sixth Annual Meeting of the National Conference on the Use of On-Line Computers in Psychology, St. Louis, November 10, 1976.

Figure 1. A "typical" timing system.

Early single-user systems, such as PSYCLE, often had an external clock that would interrupt the program after a preset interval, rather than generate interrupts at a fixed rate. In a FORTRAN framework, we can easily implement either a "typical" or "preferred" timing system with this type of clock. In the "typical" approach, DELAY sets the external clock and suspends the explicit program while waiting for the clock to strike, though it may continue background activity (e.g., emptying a Teletype buffer). A program using DELAY is shown in Table 2. It is obvious that, since subroutine SELECT consumes at least 100 msec, this program cannot maintain synchronization of this task. The selected light stays on for 500 msec, but the time between offset and onset varies from 600 to 700 msec, which is too long and unpredictable. The problem is that timing in this program is an alternative activity to CPU activity, not a parallel one. The alternative approach, shown in Table 3, uses subroutines PRESET(N), to set

the clock for N msec, and IDLE, to wait for the clock to expire. This example is a direct implementation of a "preferred" system. As IDLE invariably precedes PRESET, the two commands can easily be combined into a single system call.

In a system with a clock interrupting at a fixed rate, either line frequency (16.7 msec) or every 10 msec (as we will assume in these examples), the user cannot preset the hardware clock for a future interval, but he/she may be able to simulate such a clock with software. The counting of pulses simply is moved from an external register to an interrupt-handling routine, while the program continues to execute between these nonsignificant interrupts. On a system with more typical software, only the CALL DELAY(N) command will be found, suspending program execution for N/10 clock inter-

Figure 2. A "preferred" timing system.

Table 1
A Vocabulary of Real-Time FORTRAN Routines

SELECT(L)	Set L equal to the serial number of the next stimulus. This routine takes a variable time from 100 to 200 msec.
LITON(L)	Turn on stimulus light L.
LITOFF(L)	Turn off stimulus light L.
PRESET(N)	Preset timer for N msec.
IDLE	Wait for preset timer to interrupt.
DELAY(N)	Relinquish CPU control for N msec, to within 10-msec accuracy.
GETIME(T)	Fetch the time of day into T, measured in milliseconds from midnight.
UNTIL(T)	Relinquish CPU control until time of day T, measured as above.
FINISH(N)	Relinquish CPU control until N msec from the last time the clock returned CPU control to the task.
SCHED(TNAME,TIME,PRI0)	Begin task TNAME at TIME, running at priority PRI0.
RAISE(TNAME,PRI0)	Raise priority of task TNAME to PRI0, if PRI0 exceeds its current priority.
LATEST(TIME,PR1,PR2)	Run current task at priority PR1 until TIME, then change to PR2.
GETCPU(USE,FROM)	Request use of the CPU for at most USE out of the next FROM msec.

rupts from the moment DELAY is called. As with the variable-delay clock, use of CALL DELAY with such a hardware timer may lead to timing inaccuracies. While the PRESET and IDLE combination may be more accurate in theory, it may be less convenient in practice. Users may find it unnatural to request timing intervals before executing tasks within those intervals; it may be more convenient to program as if background calculations did not consume time, and to let the operating system account for the inaccuracies of that assumption. The next system calls provide both explicit and implicit mechanisms for accounting for the CPU time consumed by background tasks.

One explicit approach, available on all of the systems investigated, is a command such as CALL GETIME(T), which returns the current time of day as variable T. Using this command, we can determine the time of day when we begin the second 500-msec interval (and again after the CPU loop) and ask for whatever delay is needed to complete 500 msec. Table 4 shows this revised program. An alternative way of accomplishing proper timing is by using the command CALL UNTIL(T),

which suspends execution until a specified time of day rather than for a specified number of ticks of the system clock. (This is similar to the ISA-standard TURNON routine.) A program using UNTIL is shown in Table 5. An advantage of this routine is that routine GETIME need be called only once. After that, all further calls can be calculated by simple addition rather than by a possibly time-consuming call to GETIME. Another useful subroutine, not standard in any system we investigated, is CALL FINISH(N), which suspends the program until N/10 clock ticks from the time control are returned after the last timing call, rather than measuring time from the end of heavy CPU activity. FINISH accomplishes a retroactive sequence of CALL PRESET and CALL IDLE. Table 6 shows a sample program using FINISH.

Of the three proposed solutions, there is only ease of programming to distinguish the CALL UNTIL approach from the CALL FINISH one. Both are somewhat preferable to the combination of CALL GETIME and CALL DELAY. Using that combination, it is always possible for the system clock to interrupt at some point during the execution of the pair of calls, so that an extra 10 msec is added to the desired delay. This is particularly likely to happen if time of day is not a simple integer but is, instead, a vector (day, hour,

Table 2
Unsuccessful FORTRAN Program Using DELAY Alone

```

CALL SELECT(L)
DO 1 I=1,50
CALL LITON(L)
CALL DELAY(500)
CALL LITOFF(L)
CALL SELECT(L)
C TIMING ERROR OCCURS WHEN DURATION OF SELECT
C IS ADDED TO 500 MSEC INTERVAL
CALL DELAY(500)
1 CONTINUE
STOP
END

```

Table 3
Successful FORTRAN Program Using PRESET and IDLE

```

CALL SELECT(L)
C REQUEST QUICK INTERVAL TO INITIALIZE TIMER
CALL PRESET(10)
CALL IDLE
DO 1 I=1,50
CALL PRESET(500)
CALL LITON(L)
CALL IDLE
CALL PRESET(500)
CALL LITOFF(L)
C CLOCK RUNS WHILE SELECT IS CALLED
CALL SELECT(L)
CALL IDLE
1 CONTINUE
STOP
END

```

Table 4

Successful FORTRAN Program Using DELAY With GETIME

```

CALL SELECT(L)
DO 1 I=1,50
CALL LITON(L)
CALL DELAY(500)
CALL GETIME(T1)
CALL LITOFF(L)
CALL SELECT(L)
CALL GETIME(T2)
C REQUEST ONLY UNELAPSED PORTION OF 500 MSEC
CALL DELAY(500-(T2-T1))
1 CONTINUE
STOP
END

```

Table 5

Successful FORTRAN Program Using UNTIL With GETIME

```

CALL SELECT(L)
C FIND STARTING TIME FOR WHOLE EXPERIMENT
CALL GETIME(T)
DO 1 I=1,50
CALL LITON(L)
T=T+500
CALL UNTIL(T)
CALL LITOFF(L)
CALL SELECT(L)
T=T+500
C VALUE OF T NOT AFFECTED BY DURATION OF SELECT
CALL UNTIL(T)
1 CONTINUE
STOP
END

```

minute,second,millisecond), such as occurs in some systems. The conversions back and forth from this notation to a quickly updatable internal clock will, on these systems, lead to an appreciable probability of this kind of timing error. Clever operating system programming can prevent this difficulty by subtracting one clock tick from any DELAY request that occurs in the 10-msec interval after a GETIME request. But such cleverness is not likely to be a part of commercial operating systems. For that reason, either of the other two strategies is generally preferred.

MULTIPLE TASKS: PROBLEMS AND SOLUTIONS

The routines discussed so far are adequate to provide accurate timing of one experiment, but they may lead to difficulty in a multiprogramming environment that is priority driven rather than time sliced. While a time-sliced environment guarantees each program at least a fixed fraction of, say, every 25-msec interval, a priority system gives the highest priority task complete control of the CPU until it relinquishes control to wait for the clock or for I/O to complete. The operator or calling program establishes the priority for each task when it enters the system, and in some systems a task may alter its own priority or another task's priority. Imagine that we have two separate versions of our benchmark light-

driving task running, with TASK TWO 50 msec behind TASK ONE in phase. With higher numbers denoting higher priority, the two tasks have priorities 20 and 15. Figure 3 shows the intended timing diagram, and also shows what happens when priority 15 TASK TWO needs control of the processor to perform some I/O. It cannot get the control, because the priority 20 task is engaged in a heavy CPU loop. While TASK ONE is often idle, not making demands on the CPU, TASK TWO is sometimes suspended by TASK ONE, ready to execute but unable to use the processor. In particular, the TASK TWO execution of LITOFF is delayed by around 100 msec.

There is more than enough total CPU time available for both tasks, even if both calls to SELECT need to occur within an identical 500-msec interval. The problem is that the system does not distinguish between a task's I/O priority and its CPU priority, which may not be the same. While the I/O must occur as soon after a significant clock interrupt as possible, much of the CPU activity need only occur before the next significant clock tick. The task can easily relinquish the processor for other tasks' urgent I/O needs, providing that it can regain the processor at least 200 msec before it needs to turn on the next light. Clearly, what is needed is a way to separate the I/O and CPU priorities of a task.

The most direct way to separate these priorities is with a command such as CALL CHGLEV(P), which changes a task's own priority to level P. Table 7 shows the revised FORTRAN program for the two tasks, using the CALL FINISH timing method with CHGLEV, and Figure 4 shows the effect on the execution of LITON, LITOFF, and SELECT. In the new code, the urgent I/O for the TASK ONE light occurs at priority 20 in both halves of the cycle, but in the second half, TASK ONE reduces its priority to 10 before calling SELECT. At this point, SELECT can be interrupted, if necessary, by the brief but urgent I/O request of TASK TWO, running at priority 15. Once that I/O is finished, TASK TWO reverts to priority 5, allowing TASK ONE to finish

Table 6

Successful FORTRAN Program Using FINISH Call

```

CALL SELECT(L)
C STALL BRIEFLY TO ESTABLISH REFERENCE POINT
C FOR FIRST CALL TO SUBROUTINE FINISH
CALL DELAY(10)
DO 1 I=1,50
CALL LITON(L)
CALL FINISH(500)
C TIME IS "THEN"
CALL LITOFF(L)
CALL SELECT(L)
C TIME IS "NOW"
C MEASURE 500 MSEC FROM "THEN", NOT FROM "NOW"
CALL FINISH(500)
1 CONTINUE
STOP
END

```

Figure 3. (a) TASK ONE runs as scheduled. (b) TASK TWO is scheduled to run as shown. (c) CPU contention delays TASK TWO to this schedule.

SELECT before reasserting priority 20 and suspending execution until the next significant clock tick. With TASK ONE idle, priority 5 is sufficient for TASK TWO to gain processor control to finish its call to SELECT before the results are again needed. A similar scheme could clearly be implemented for more than two tasks.²

A difficulty with the proposed priority scheme is that it presupposes a high degree of cooperation among tasks. Any process that intentionally or inadvertently stays at a high priority during lengthy CPU activity will inhibit other tasks from doing their I/O. One obvious problem is that of low-priority users submitting high-priority jobs. This problem gets into issues of system security and is somewhat outside the scope of this paper. What we will consider instead is the case in which a single user or system manager is willing, if necessary, to sacrifice timing on a low-priority task in favor of a high-priority task when there is a shortage of total CPU time, but would prefer to have both tasks maintain their synchronizations whenever possible. One solution is an operating system that prevents any task from staying at high priority for more than one clock tick, just enough time to accomplish urgent I/O and trivial CPU activity. This solution cannot be implemented from FORTRAN. It requires extensive rewriting of the operating system.

Another solution is potentially available at the FORTRAN level, and this solution can protect a vital program from being inhibited by a less vital one. This solution involves the use of some new system commands listed in Table 1. Command SCHED(TNAME, TIME, PRIO) schedules the task called TNAME to begin execution at TIME, with priority PRIO. (This is similar to ISA-standard routine START, with the addition of a priority specification.) Command RAISE(TNAME, PRIO) immediately raises the priority of task TNAME to the maximum of PRIO and its current priority. Using these commands, we can split the highest priority task into

two parts: one part to do the actual I/O and CPU work, the other to insure that the first part gets the priority it needs. Table 8 shows these two parts, which the operating system knows as tasks RUN and MONITR. At priority 20 RUN turns on light L and reschedules itself to resume 500 msec later. Since there are no CPU tasks to be performed in the meantime, no other task will be inhibited by this high priority. As soon as it has turned off the light, RUN schedules MONITR to execute at priority 20, starting 300 msec later, then reschedules itself for priority 10. For the next 300 msec, subroutine SELECT of RUN will execute if the CPU is otherwise free, but will relinquish control to higher priority tasks if they need it. But once there are only 200 msec left to select the next stimulus, RUN should have priority over all other tasks if it needs that priority, so its priority is raised to 20 again by MONITR. When the call to SELECT is completed, RUN schedules itself to turn on light L at the appropriate time. Figure 4 continues to depict the timing of the TASK ONE and TASK TWO under this new coding and under the further modifications to be discussed later.

The combination of RUN and its associated MONITR

Table 7
Two Tasks Cooperating to Let Both Tasks' I/O Activity Take Precedence Over Both Tasks' CPU Activity

C	TASK ONE
	CALL SELECT(L)
	CALL CHGLEV(20)
	DO 1 I=1,50
	CALL LITON(L)
	CALL FINISH(500)
	CALL LITOFF(L)
	CALL CHGLEV(10)
C	RUN SELECT AT LOW PRIORITY
	CALL SELECT(L)
C	BUT RESPOND AT HIGH PRIORITY TO CLOSE LOOP
C	AND TURN ON NEXT SIGNAL LIGHT
	CALL CHGLEV(20)
	CALL FINISH(500)
1	CONTINUE
	STOP
	END
C	TASK TWO
	CALL SELECT(L)
	CALL CHGLEV(15)
	DO 1 I=1,50
	CALL LITON(L)
	CALL FINISH(500)
	CALL LITOFF(L)
	CALL CHGLEV(5)
C	RUN SELECT AT LOW PRIORITY
	CALL SELECT(L)
C	BUT RESPOND AT HIGH PRIORITY TO CLOSE LOOP
C	AND TURN ON NEXT SIGNAL LIGHT
	CALL CHGLEV(15)
	CALL FINISH(500)
1	CONTINUE
	STOP
	END

Figure 4. TASK ONE allows TASK TWO to interrupt background CPU activity and perform urgent I/O tasks.

is just about what is needed. If the CPU is otherwise free, RUN can finish its call to SELECT at a leisurely pace and thus leave the CPU free for any tasks that need it during the last 300 msec of the 500-msec light-off interval. On the other hand, even if the CPU is busy for the beginning of that interval, RUN can regain control in time to complete its tasks by the end of the interval. There is only one immediately apparent improvement in this split task approach: If subroutine SELECT can somehow determine that it is half finished with its task and 300 msec of the interval have not yet elapsed, it can reschedule MONITR for 100 msec later than otherwise, thus freeing an additional 100 msec of CPU time for sharing with tasks of lower I/O priority. Further dynamic subdivisions of the 200-msec estimate minimize the effect on other tasks.

PROPOSALS FOR FUTURE SOLUTIONS

The division of a single task into RUN and MONITR is within the capability of some commercial FORTRAN systems we have investigated, but is still not an ideal solution to the scheduling problem. In order to maintain the synchronization of each task with minimal impact on other programs, we have split it into two tasks. However, the protocol for the task is really not that of a multitasking situation, where two separate I/O protocols run in asynchronous parallel or where I/O events must interrupt ongoing CPU activity. (For an example of the latter situation, imagine that the selected light must be flashed on and off every 50 msec while the subsequent light is being selected.) Instead, it is a single execution stream with a mixture of urgent and delayable tasks that must, nonetheless, be performed in strict sequence. Thus, an ideal operating system would allow any task to be synchronized without the separate compilation of an auxiliary program MONITR.

While the usual, commercially available operating system allows the user to specify when a given task should begin, what is often needed is one that allows

the user to specify when some tasks begin and the latest time that other tasks should end. Subroutine LATEST provides one mechanism for this specification: executing CALL LATEST(TIME,PRIO1,PRIO2) would change the current priority of the program to PRIO1 until TIME, then raise it to PRIO2. This essentially moves the MONITR from another task into the heart of the operating system, where it executes with less task-switching overhead. If the calling program sets TIME to be the next significant clock tick minus the expected duration of the intervening CPU activity, then this effectively schedules the call to SELECT by its latest allowable ending time rather than by any fixed beginning time. However, without dynamic updating of the 200-msec estimate for SELECT, there is still the possibility that a half-completed task would demand the reinstatement of its high CPU priority before that high priority was absolutely indispensable, and thus create some unnecessary interference with other tasks. Another minor problem is that we are forced to select two explicit priorities for a single task. The program in Table 9 demonstrates the use of LATEST as another way of implementing the priority change of the previous example.

Every 10 msec the system clock ticks, and every 10 msec the system should know which tasks have been executing during the last 10 msec. Rather than requiring individual tasks to declare that they are half done, three-quarters done, and so on, it would be more efficient for the operating system to simply keep track of the CPU time consumed by tasks. Taking this approach, we arrive at the final proposed system utility: CALL GETCPU(USE,FROM) requests the control of the CPU for background computation for a maximum of USE msec out of the next FROM msec. Note that there is no need to specify any secondary or background priority in this command; it is simply assumed that this

Table 8
A Task With a Separate Task Monitor to Reestablish Its Priority

```

TASK RUN
CALL SELECT(L)
CALL CHGLEV(20)
DO 1 I=1,50
CALL LITON(L)
CALL FINISH(500)
CALL LITOFF(L)
CALL SCHED('MONITR',T+300,20)
CALL CHGLEV(10)
CALL SELECT(L)
CALL CHGLEV(20)
CALL FINISH(500)
1 CONTINUE
STOP
END

TASK MONITR
CALL RAISE('RUN',20)
STOP
END

```

Table 9
A Task Using LATEST to Schedule Its Change of Priority

```

C   START AT PRIORITY 20
    CALL SELECT(L)
    CALL DELAY(10)
    DO 1 I=1,50
    CALL LITON(L)
    CALL FINISH(500)
    CALL LITOFF(L)
C   CHANGE TO PRIORITY 10 NOW,
C   BACK TO 20 IN 300 MSEC
    CALL LATEST(300,10,20)
    CALL FINISH(500)
1  CONTINUE
    STOP
    END

```

CPU time is needed to accomplish forthcoming I/O that has a preestablished task priority. This call would alternate with calls to UNTIL or FINISH, which request exact time scheduling for I/O activity and any very brief, associated CPU activity needed to conduct or record that I/O. Table 10 shows this command being used to schedule the basic benchmark program.

Using the GETCPU command, the operating system needs to keep a constantly updated list of CPU time requests by deadlines for obtaining that CPU time, and needs to subtract consumed CPU time from current requests as the clock interrupts those tasks that have CPU control. If the available time forthcoming is enough for all tasks, then control is given to the task with the earliest deadline, even if that task has the lowest priority. The priority structure comes into effect only when the predicted demand for CPU time exceeds the supply. While this places a greater decision load on the operating system than does a standard priority system, it also maximizes the number of tasks that maintain their synchronizations while still giving the best service to the highest priority tasks. While an operating system with these features may not now be commercially available, such a scheduling algorithm is not beyond our present capacity to implement.

COPING WITH REAL-TIME FORTRAN

While it is rather obvious that CALL SELECT will require something more sophisticated than CALL DELAY following it, it may not be so obvious that the same problem may also arise in the first half of the basic benchmark program. Here we have assumed that the duration of LITON is negligible. In practice, even such seemingly innocuous commands as LITON or moving four characters to an output I/O buffer may consume more than the 10-msec interval between clock interrupts. In the case of character output, the FORTRAN I/O handler may not return control to the calling program until all characters generated have been output, or else the interrupts generated by the printer as it is ready for each new character may take priority

over the supposedly foreground program. We have encountered both of these problems while investigating multiuser systems. Multiuser systems also have a further problem: To maintain protection between tasks, all I/O require calling the system supervisor, which must switch context from user to executive mode, test the legality of the request, initiate the I/O (which may involve inserting the request in a complex queue of such requests), reestablish user mode context, and return to the calling program. Such a sequence of events may well enhance total throughput on a multiuser system, but tends to reduce or cancel the beneficial effects of buffering on any particular critically timed program. As a result, the use of DELAY is generally inadequate after any task requiring the intervention of the system executive.

It is obvious that I/O requests require such intervention. It may be less obvious that getting the time of day, calling reentrant subroutines in otherwise protected memory, or requesting the initiation of another task may be similarly time consuming. This leaves only arithmetic as a function local to the user program, and there is no reason to time-schedule arithmetic without scheduling some associated I/O or time-measuring activity. I would suggest that using DELAY without GETIME has an unacceptably high probability of failing to produce the nominal 10-msec synchronization of a typical system. Combining DELAY with GETIME is a major improvement but not a total solution; it requires two or three system calls per timed interval, rather than one. Many systems offer only DELAY and GETIME in their standard vocabulary, so perhaps the best compromise is a user-written FINISH routine, where FINISH itself handles the necessary calls to GETIME and DELAY. As the overhead of the multiple calls is unavoidable, this simplifies the coding of the main program.

There is an important ambiguity in the specification of FINISH which suspends the program "from the time

Table 10
A Task Using GETCPU to Allow the Operating System to Decide When to Give It Control for Background Computation

```

    CALL SELECT(L)
C   TASK NORMALLY RUNS AT HIGH PRIORITY
    CALL DELAY(10)
    DO 1 I=1,50
    CALL LITON(L)
    CALL FINISH(500)
    CALL LITOFF(L)
C   THESE 200 MSEC OF CPU TIME ARE NOT NEEDED
C   IMMEDIATELY, BUT ARE NEEDED BEFORE 500 MSEC
C   HAVE ELAPSED
    CALL GETCPU(200,500)
    CALL SELECT(L)
C   REASSERT PRIORITY
    CALL FINISH(500)
1  CONTINUE
    STOP
    END

```

control was returned after the last timing call." Especially in a multiple-user system, this can be interpreted as either the time control should have been returned, that is, the program was placed on a "ready-to-execute" list, or else the time CPU activity actually resumed. A similar problem arises with any of the other timing calls, that is, how do the calls indicate whether timing was accurate or delayed? The one version of ISA-standard WAIT that I have seen implemented includes an error parameter for illegal or impossible wait requests, but no parameter that indicates the system's inability to meet a perfectly legal request, due to competition from other tasks.

In FORTRAN syntax, perhaps the simplest way to pass timing success information back to the calling program is to change these subroutines to functions. For example, the value of function CALL FINISH(N) would be the delay in milliseconds between the time that CPU activity was scheduled to resume and the time it actually resumed. Thus, a statement such as

```
IF (FINISH(500).GT.20) GO TO 100
```

simultaneously requests the completion of a 500-msec interval and provides an error-branch facility if synchronization is delayed by more than 20 msec. By using the GETIME routine, a user-written FINISH function

can easily provide this kind of timing information to the calling task. This would then be free to use or ignore the information, depending on the precision with which the task needed to be scheduled.

REFERENCES

- CASTELLAN, N. J. Standards: Real-time extensions to programming systems. *Behavior Research Methods & Instrumentation*, 1976, 8, 207-210.
- CREELMAN, C. D. Software management of timing in computer-controlled on-line experiments. *Behavior Research Methods & Instrumentation*, 1974, 6, 488-492.

NOTES

1. I tried to find a copy of these standards at both the Engineering Library and the Computer Research Facility of the University of Toronto, and was unsuccessful. I realize that I am generalizing from a very small sample, but I think that the difficulty of locating the standards at two such likely places suggests that they are not commonly available and are not widely followed in the computing community. My discussion is based on two secondary sources: the Castellan article and a description of the ISA-standard FORTRAN extensions provided by a vendor whose computer we were considering.

2. If the CHGLEV subroutine is not available on the system, one alternative may be to schedule SELECT as a separate task, running at priority 10 or 5 while the calling task runs at priority 20 or 15.