Some observations on 'least time to go' scheduling

J. F. Lubran and J. D. Roberts
University Engineering Department, Cambridge

The mechanism of scheduling time critical programs by a 'least time to go' algorithm is shown to provide an attractively simple and secure facility for use by object programs which is practicable to implement. It is also shown to possess advantages of robustness as well as optimality.
(Received December 1970)

In real time applications of computers such as hybrid computation and process control, a single processor may be required to execute several time critical processes. A typical single process will include short highly time critical phases such as the operation of ADC's or DAC's on an analog computer. These phases of duration δT_1, $\delta T_2, \ldots$ must be performed in immediate response to events at times $T_1, T_2, \ldots$. A process may also include intermediate phases, such as the computation of the next values to be written to the DAC's, which must be executed at any time within the intervals $(T_1, T_2), (T_2, T_3) \ldots$. For the purpose of this description a 'crisis' as discussed by Middleton (1971) is considered to be an extra event which trggiers a null phase of computation. This paper considers the problem of scheduling several time critical processes in parallel. The premises on which the present discussion is based are given below. They perhaps apply less equivocally to hybrid computation in which there are well-defined quantitative accuracy considerations than to other real time applications.

1. There is no problem of overlap between the different processes in the highly time critical intervals $(T_i, T_i + \delta T_i)$. For this condition, *either* the interval δT_i must be negligible *or* the timing must have been so planned that no event associated with another process occurs in the interval $(T_i, T_i + \delta T_i)$. The latter condition is easy to fulfil deliberately in hybrid computation.
2. An intermediate time critical computation phase may be distributed in any way with equal effectiveness over the interval (T_i, T_{i+1}). The only criterion is reliability in completing it before T_{i+1}.
3. The event times and amounts of computation are so distributed as to make it possible for some scheduling decision to fulfil criterion (2).
4. Subject to conditions (1) and (3) events may occur at non-tabular intervals.

A common simple scheduling scheme associates each process with a priority 'level', this notion being embodied in the interrupt processing hardware of many computers. The need to generalise this scheme is indicated in the BCS Report (1967) which proposes that priorities be manipulated dynamically by the user. A difficulty in this approach is that to allow sufficient privilege to schedule effectively also allows sufficient facility for processes to interact completely arbitrarily and to generate unreproducible effects.

An alternative method of scheduling is to work in terms of time to go rather than priority. A simple but effective scheme is to run the process associated with the earliest next event out of those processes which have yet to complete their current phases of computation. The conceptual simplicity of this scheme has been emphasised by Fineberg and Serlin (1967) with reference to the problem of running two or more time critical computations simultaneously on a single processor. These authors advocate that in hybrid computation multi-level interrupt hardware is 'highly overrated'. In 'least time to go' scheduling the only link which a process need make with the scheduler is to give information about the urgency of its next event time. In this way the composition of a real time program does not involve explicit references from within a process to different processes. It resembles more closely the implementation on a set of dedicated processors. Another advantage of this methodis that alterations of priorities take place only in response to events, and not at undefined times. Programs which achieve effective scheduling should not need to be complicated and even erroneous design of a program will not produce the same degree of chaos as could be produced by completely arbitrary manipulation of priorities. Note that it is the time before the next event that determines the urgency and not the process run time, therefore the application is not impeded by ignorance of fluctuations of process run time.

A statement and proof of the optimality of this kind of scheduling (but using a different terminology) is given by Conway, Maxwell, and Miller (1967). Referring to a management sciences report of Jackson (1955) they state that 'The maximum job lateness (the negative of the closest margin before a deadline) . . . is minimised by sequencing jobs in order of non-decreasing due-dates ('least time to go' scheduling)'.

Sometimes properties of particular scheduling strategies are exploited for purposes other than scheduling. For example, fixed priority structures can be used to mutually exclude processes at the same level from use of the same resource. This is more selective than complete inhibition of interrupts but less selective than a more general semaphore system. Since advances in hardware and software design are being made which enable the latter completely selective kind of system to be implemented efficiently, the authors feel that scheduling methods should be assessed for their properties as schedulers alone.

These considerations present a strong prima facie case for incorporating 'least time to go' scheduling at a fundamental level in a real time computing system. It may be that the reason why it has not received more widespread interest is that it is considered to be inefficient or to possess concealed disadvantages. It is the purpose of this paper to show that:

1. 'Least time to go' scheduling can be reasonably efficient on present day computers with good multiprogramming hardware.
2. 'Least time to go' scheduling possesses less obvious advantages in its robustness to various types of perturbation. These could include delays caused by higher priority interrupts outside the scheme of 'least time to go' scheduling.

Implementation

Each process must be in one of the following states:

(A) waiting for a particular event to occur
(B) running indivisibly in immediate response to an event

Downloaded from https://academic.oup.com/comjnl/article/15/1/32/418374 by guest on 19 April 2024

(C) (i) running during an intermediate computation phase
(ii) ready to run but awaiting completion of a more urgent phase of another process.

Transitions from phase A to phase B are controlled by interrupts signalling the occurrence of events. Transitions from B to C and from C to A are controlled by instructions in the code defining each process. In any one process these transitions must occur in the cyclic sequence A B C viz:

phase (A)
 interrupt (<event>)
phase (B)
 continue (<time to go>)
phase (C)
 await (<event>)
phase (A)

If any event occurs before the corresponding 'await' statement has been executed, it means that the programmer has either specified his time to go incorrectly or that he is demanding more computing power than is available.

The information required to resume a process is stored in a vector of contiguous locations reserved for that process. Each vector also contains information about the urgency (event time) and a pointer which can be used to chain the processes together in a queue. Each process in phase (A) is linked to a pointer in an interrupt table specially reserved for the appropriate event. A process in phase (B) is linked to the 'current process pointer'. All other processes are chained together in a one-way queue in order of urgency. The most urgent process is linked to the 'head process pointer' which takes the same value as the 'current process pointer' when there is no process in phase (B). The least urgent process is always the 'idle loop' with infinite event time.

The function of the scheduler is to maintain the configuration of linked vectors on each transition. When an interrupt occurs the process linked to the appropriate element of the interrupt table becomes the current process. This element of the interrupt table is then cleared. On the 'continue' instruction, a search is made down the chain of active processes and the current process is inserted before the first process with a later event time. The head of the chain becomes the current process. On the 'await' instruction the current process which is also the 'head' process is removed from the chain of active processes and linked to the appropriate pointer in the interrupt table.

The only action presenting any problem of speed is the 'continue' instruction which involves a search down a chain of vectors with interrupts inhibited. The following favourable points must be taken into account here:

1. The most frequent event will typically be those with short computations to be performed with great urgency and these will be inserted near the head of the chain of active processes.
2. The 'continue' instruction will take place at a short time after the highly time critical interrupt routine has been completed. There is no danger of coincidence with other time critical events if condition (1) of the introduction is satisfied.

The precise algorithm is conveniently expressed in the record handling notation of Wirth and Hoare (1966); although it is emphasised that no automatic translator was used in implementation. In fact, many of the functions are incorporated within the hardware of the computer which was used. The process reference 'current' is represented by a certain location which is built into the hardware of the machine. The procedures 'restore hardware registers' and 'reactivate current process' are combined in a single hardware function.

```
integer procedure time now†; comment time measured in 1/1024
secs;
record class process;
begin integer array hardware registers;
      integer event time;
      ref (process) successor
end;
ref (process) current†, head;
procedure preserve hardware registers; comment the hardware
registers in the current process vector are set from the actual
hardware registers in the central processor;
procedure restore hardware registers†; comment the reverse of
the above;
procedure reactivate current process†;
procedure alarm; comment in hybrid computing it is useful to
put the analog into 'hold' mode and in any case to give a
warning such as ringing the bell on the typewriter;
procedure await (integer value event class);
begin table [event class] := current;
      current := head := successor (current)
end;
procedure interrupt (integer value event class);
begin preserve hardware registers;
       if table [event class] = null then alarm else
       begin current := table [event class];
           table [event class] := null;
           reactivate current process
       end
end;
procedure continue (integer value time to go);
begin integer t;
      t := event time (current) := time to go + time now†;
      queue (t)
end;
procedure queue (integer value t);
if t < event time (head)
then begin successor (current) := head;
           head := current;
           reactivate current process
     end
else begin ref (process) last, next;
           last := head; next := successor (head);
           while not t ≤ event time (next) do
                 begin last := next;
                       next := successor (last)
                 end;
           successor (last) := current;
           successor (current) := next;
           current := head; restore hardware registers; re-
           activate current process;
     end;
```

On an ICL/Elliott 4130 computer* with a 2 μsec core store, timing for the various operations is as follows:

switch to executive mode (preserving hardware registers)	20 μsec	(hardware)
determine interrupt event class	37 to 84 μsec	(has to be performed by software because of current nature of interrupt linkage)
setting current pointer etc.	30 μsec	
reactivation of process (restoring hardware registers)	22 μsec	
Total overhead for 'interrupt'	109 to 156 μsec	

†Denotes built in hardware functions.
*The digital part of the Cambridge University Control Group hybrid computer 'Cassandra'.

Downloaded from https://academic.oup.com/comjnl/article/15/1/32/418374 by guest on 19 April 2024

switch to executive mode (preserving hardware registers)	20 μsec
insertion of vector at top of chain	68 μsec
insertion of vector in 2nd position in chain	110 μsec
for each additional search down chain	22 μsec
reactivation of head process (restoring hardware registers)	22 μsec
Total overhead for 'continue'	110 to 152 μsec + 22 μsec × (final position in chain − 2)

switch to executive mode (preserving hardware registers)	20 μsec
link to interrupt table etc.	31 μsec
reactivation of head process (restoring hardware registers)	22 μsec
Total overhead for 'await'	73 μsec

A count is kept of the time in one of the locations in core which is incremented by a pulse generated every 1/1024 sec. In this particular computer, the count was driven by the same hardware as is used for sharing autonomous input and output of all slow devices and the overhead incurred of three core cycles per pulse amounts to less than 2/3% of the core store. The overheads listed above do not include the normal checks which are necessary to provide complete protection between different programs. If this protection is required extra times must be added equivalent to the normal overheads on trapped instructions in general purpose multiprogramming executives. Experience with the particular computer used would indicate that these two types of overhead are of comparable order of magnitude. The scheduling system described does not therefore make unreasonable demands on the processor. In any case, the only part of the scheduling system which would not be needed by a 'fixed priority' system is the searching triggered by the 'continue' instruction, and even this would appear to be unlikely to dominate the other parts.

It should be noted that the above algorithm works in absolute times and that a 24-bit integer allows counting for 5 hours without overflow. With shorter word length or longer working, the algorithm could be modified to work in times relative to the previous event.

Robustness of 'least time to go'

An example given by Fineberg and Serlin (1967) illustrates the ability of 'least time to go' scheduling to make full use of processing power by avoiding idling. Here two tasks require equal proportions of processor time in regular intervals in the ratio 3:2. While 'fixed priority' scheduling would allow only 6/7 of the processor time to be used, 'least time to go' scheduling allows 100% processor utilisation. **Fig. 1** shows that an attempt to achieve 100% processor loading under fixed priority scheduling results in one process failing to meet its deadline. **Fig. 2** shows how 'least time to go' scheduling achieves 100% processor loading without lateness. The ordinate of each graph indicates the amount of uncompleted computation before the next event. The gradient is zero when a process is suspended and −1 when actually running.

If the critical times of the two processes do not coincide, then the utilisation of the processor obtainable with 'fixed priority' scheduling may be bigger than 6/7. It may even approach 100% when there is the maximum distance of one quarter of the shorter period between critical times associated with the two processes as shown in **Fig. 3**. Under these conditions, however,

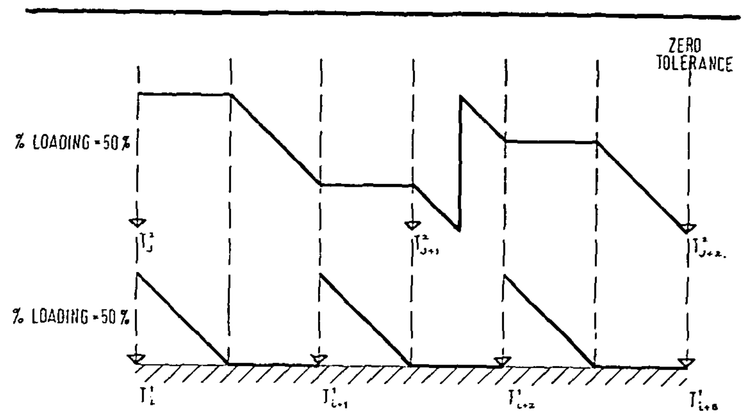

Fig. 1. Fixed priority with coincident events

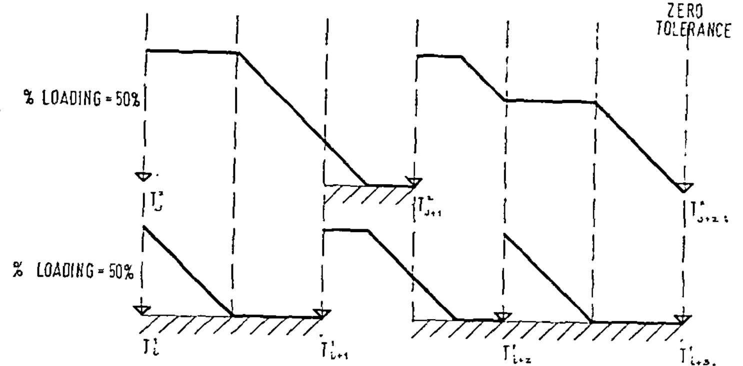

Fig. 2. Least time to go with coincident events

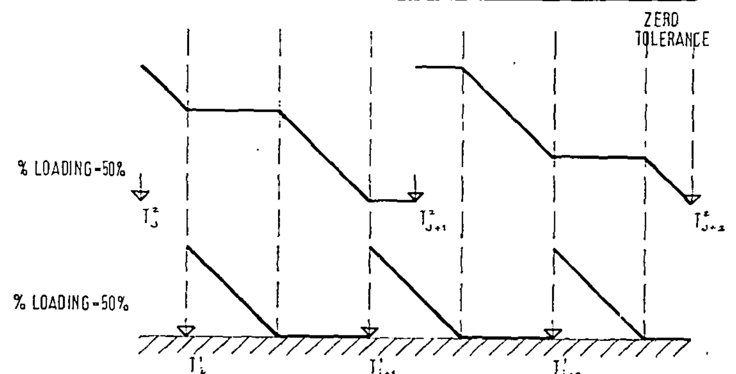

Fig. 3. Fixed priority with staggered events

'least time to go' scheduling possesses more subtle advantages. These arise when one takes into account tolerances in the time by which phases of computation are completed. In a real situation, the scheduler may need to allow for any of the following hazards:

1. Short higher priority interrupts from asynchronous devices.
2. Effective degradation of processor power through loss of core cycles to higher priority I/O processors.
3. Coincidence of use of non-reenterable subroutines by parallel tasks.

In case (3) the tasks will exclude each other by a Dijkstra semaphore and the most urgent task may be kept waiting for a period up to the duration of the subroutine. In this case, time is only 'borrowed' by the environment. Even if there is continual 100% loading of the processor, the critical times are met provided that the time borrowed is within a certain tolerance. This tolerance is the minimum to spare which any process has when it becomes ready for the next time critical event (for example, as shown in **Fig. 4**). In the first two cases, time is actually 'stolen' by the environment. It can be recovered only during time which would otherwise be idle, therefore it can only be recovered if the loading is actually less than 100%.

Downloaded from https://academic.oup.com/comjnl/article/15/1/32/418374 by guest on 19 April 2024

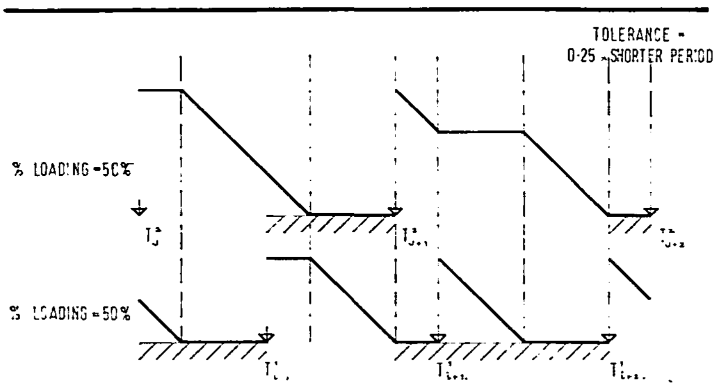

Fig. 4. Least time to go with staggered events

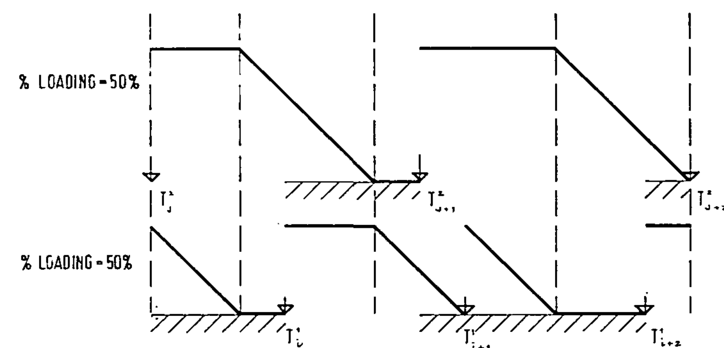

Fig. 5. Least time to go with staggered events after maximum perturbation

Table 1 Table of the maximum loading and tolerance of two processes under the 'Fixed priority' and 'Least time to go' schedulers

	'FIXED PRIORITY' SCHEDULING	'LEAST TIME TO GO' SCHEDULING
Coincident events	6/7 loading: no tolerance	100% loading: no tolerance (Fig. 2)
Staggered events	100% loading: no tolerance (Fig. 3) OR 6/7 loading: ¼ short period tolerance	100% loading: ¼ short period tolerance (Fig. 4)

Isolated demands of this type within the relevant tolerances may still nevertheless be accepted even if the loading is only very slightly less than 100%. The timing for 'least time to go' scheduling is shown in Fig. 4. Here every phase of computation is completed with at least one quarter of the shorter period to spare and this time may be 'borrowed' quite freely by the environment. The effect on the timing when the environment claims the maximum tolerance is shown in **Fig. 5.** The timing returns to that shown in Fig. 4 when the time is reclaimed assuming the loading to be slightly less than 100%. In contrast, there is no tolerance at all with fixed priorities (Fig. 3) and to obtain a tolerance of one quarter of the shorter period, the processor would again have to be only 6/7 loaded. With 'least time to go' scheduling, the tolerance interval is zero only if critical events in different processes coincide. These observations are summarised in **Table 1.** The formula for the tolerance in general, derived in the Appendix, shows that with more than two processes the tolerance is in general non-zero even if events in some of the processes coincide. There is zero tolerance only if there is a simultaneous coincidence of the time critical events of all the processes. While it may not be good practice to run the processor at 100% loading as indicated in the examples, these are extreme limits of a range of feasible loadings. For more reasonable processor loadings margins of tolerance will be correspondingly greater.

Conclusion

'Least time to go' scheduling presents an attractively simple linkage with tasks governed by time critical events. It is efficient and realistic even on some present day general purpose real time computers, and could easily be made even more efficient with hardware implementation of parts of the algorithm. It allows the maximum possible use of the computing power available and possesses less obvious but probably more important advantages in robustness and insensitivity to the kinds of perturbation to timing which occur in a multiprogramming environment. The role of fixed priorities is not completely eliminated but its significance is both reduced and clarified in the context of higher priority interrupts as one kind of perturbation. 'Least time to go' scheduling will also allow effective scheduling when events occur at irregular intervals.

Appendix

The notion of percentage loading of a processor is meaningful if for each process $P^{(j)}$ the amount of computation to be performed during each interval $\Delta T_i^{(j)}$ is $\theta^{(j)} \Delta T_i^{(j)}$ where $\Delta T_i^{(j)} = T_{i+1}^{(j)} - T_i^{(j)}$ and $\theta^{(j)}$ which represents the process loading is the same for every interval in the process $P^{(j)}$. The overall processor loading is defined by:

$$\Theta = \sum_j \theta^{(j)}$$

To obtain a compact but strong result which gives the tolerance, i.e. the minimum time spare before a critical event, remains a research problem. Nevertheless, the following results are strong in some useful cases and do demonstrate the essential point that the tolerance is always non-zero even with 100% loading provided that there is never a total coincidence of events in all the processes. The method used is to generate successive improvements on an initial strategy of 'infinitesimal time slicing' (Fineberg and Serlin, 1967). The strategy to which the results are shown to apply is intermediate between 'least time to go' and 'infinitesimal time slicing'. The results apply *a fortiori* by the theorem of Jackson (1955) discussed earlier to 'least time to go' scheduling.

Two processes

The most convenient way is to state the required result not as a simple formula but as a set of sufficient conditions which when satisfied simultaneously allow a tolerance τ to be obtained.
Condition 1:

$$\tau \leqslant \left|T_i^{(1)} - T_{i'}^{(2)}\right| \text{ all } i, i'$$

i.e. τ must be a lower bound to the smallest gap between any pair of critical times.

Any interval in a process will either (a) be nested in an interval of the other process or (b) by condition 1, overlap a period of at least τ in each of two successive intervals of the other process. For case (a) the computation is rearranged to give absolute priority to the nested phase in accordance with 'least time to go' scheduling. The state of the processes at the end of the longer interval is unchanged. The nested phase will terminate with tolerance $> \tau$ under the following condition:
Condition 2:

$$\tau \leqslant (1 - \theta^{(j)}) \Delta T_i^{(j)} \quad T_{i'}^{(j')} < T_i^{(j)} < T_{i+1}^{(j)} < T_{i'+1}^{(j')}$$

For case (b), consider an initial strategy of 'infinitesimal time slicing' with processor power allocated to each of the processes of $\theta^{(1)}/\Theta$ and $\theta^{(2)}/\Theta$. Any tolerance $\tau^{(2)}$ in an interval $\Delta T_i^{(1)}$ can be shown to propagate a tolerance $\tau^{(2)}$ in the interval in $P^{(2)}$ overlapping the end of $\Delta T_i^{(1)}$. If in the interval $\Delta T_i^{(1)}$ the

Downloaded from https://academic.oup.com/comjnl/article/15/1/32/418374 by guest on 19 April 2024

time slicing strategy is further modified to include a 'least time to go' strategy for the end of the interval, then a tolerance $\tau^{(1)}$ will be produced for that interval. Condition 1 implies that if $\tau^{(1)} < \tau$ then a quantity $(\theta^{(1)}/\Theta)\,\tau^{(1)}$ of processor activity at the end of $\Delta T_i^{(1)}$ is freed for utilisation by process $P^{(2)}$. It is easily shown that a tolerance $\tau^{(2)}$ is allowed in the time to spare before the next event in $P^{(2)}$ if

$$\tau^{(2)} \leqslant \min\left(\tau, \frac{\theta^{(1)}}{\theta^{(2)}}\tau^{(1)} + (1 - \Theta)\,\Delta T_i^{(2)}\right)$$

This may be summarised by
Condition 3:

$$\exists \tau^{(1)}, \tau^{(2)} \leqslant \tau$$

such that

$$\tau^{(2)} \leqslant \frac{\theta^{(1)}}{\theta^{(2)}}\tau^{(1)} + (1 - \Theta)\min_i \Delta T_i^{(2)}$$

$$\tau^{(1)} \leqslant \frac{\theta^{(2)}}{\theta^{(1)}}\tau^{(2)} + (1 - \Theta)\min_i \Delta T_i^{(1)}$$

A more compact but stronger condition is
Condition 3*a*:

$$\tau(1 - \min\left(\frac{\theta^{(1)}}{\theta^{(2)}}, \frac{\theta^{(2)}}{\theta^{(1)}}\right) \leqslant (1 - \Theta)\min_i \Delta T_i)$$

Conditions 1, 2 and 3a are adequate to show that the set-up in which $\theta^{(1)} = \theta^{(2)} = \frac{1}{2}$ and $\Delta T^{(1)}:\Delta T^{(2)} = 3:2$ with events maximally staggered leads to a tolerance of one sixth of the longer period.

More than two processes

With a greater number of processes ($n > 2$) much more complex combinations of possibilities exist. The problem may be regarded as involving combinations of $n(n - 1)/2$ pairs of processes, each pair using a share $2/n(n - 1)$ of the total processor power. The argument implied is again based on systematic development of a strategy starting with 'infinitesimal time slicing'. The tolerance τ is at least that time in any interval which can be saved by some pair of processes out of its limited power $2/n(n - 1)$. A weak but easily derived set of conditions is therefore as follows:

Condition 1*:

$$\tau \leqslant \frac{2}{n(n-1)} \max_{j'} \left|T_i^{(j)} - T_{i'}^{(j')}\right| \text{ all } j, i, i'$$

Condition 2*:

$$\tau \leqslant \frac{2}{n(n-1)}(1 - \theta^{(j)})\,\Delta T_i^{(j)} \quad T_{i'}^{(j')} < T_i^{(j)} < T_{i+1}^{(j)} < T_{i'+1}^{(j')}$$

Condition 3*a**: $\exists j, j'$
such that

$$\tau\left(1 - \min\left(\frac{\theta^{(j)}}{\theta^{(j')}}, \frac{\theta^{(j')}}{\theta^{(j)}}\right)\right) \leqslant \frac{2}{n(n-1)}(1 - \Theta)\min_i \Delta T_i$$

References

BCS Specialist Group (1967). A language for real-time systems, *The Computer Bulletin*, Vol. 11, No. 3, pp. 202-212.

Conway, R. W., Maxwell, W. L., and Miller, L. W. (1967). *Theory of scheduling*, Addison-Wesley Publishing Company.

Fineberg, M. S., and Serlin, O. (1967). Multiprogramming for Hybrid computation, Proc. AFIPS, Fall Joint Computer Conference.

Jackson, J. R. (January, 1955). Scheduling a production line to minimise maximum tardiness, Research Report 43, Management Sciences Research Project, U.C.L.A.

Middleton, M. D. (March, 1971). A Note on Scheduling Real Time Processes, Cambridge University Engineering Department.

Wirth, N., and Hoare, C. A. R. (1966). A contribution to the development of Algol, *CACM*, Vol. 9, No. 6, pp. 413-31.

Correspondence

(Continued from page 4)

The basic principles behind these examples are two:

1. A programming language should include much *redundancy*, which means that when a correct program is erroneously modified, then as many of the logically false modifications as possible should introduce not only logical errors but also language errors into the program.
2. When a programming language construct is *ambiguous*, that is can be interpreted in more than one way (like $a := i$ in the example above), then the whole construct should be forbidden, forcing the programmer to use other, unambiguous constructs instead (like $a :=$ **round** i in ALGOL 68).

These two principles have one common aspect: The larger ability to detect errors for the compiler is gained by restricting the freedom of the programmer to write anything he likes and have the compiler try to understand what he means. Such an *understanding* compiler is very dangerous, since it will not be as good at detecting logical errors.

This is the basic conflict which the authors of the paper in your February issue did not discuss. The authors want to change high level languages to be more *understanding*. But the disadvantages with such changes are sometimes much larger than their advantages. Especially with the advent of time-shared computers, the correction of compiler-detected errors becomes very simple. The real risk is not the compiler-detected errors, as the authors seem to think, but the logical errors not discovered by the compiler.

Yours faithfully,
J. Palme

Datalogy Section
Research Institute of National Defense
S-10450 Stockholm 80
Sweden
16 August, 1971

References

Palme, J. (1969). What is a good programming language? FOA P Report C 8231. Research Institute of National Defense, S-10450 Stockholm 80, Sweden.

Palme, J. (1971). Simula 67—An advanced programming and simulation language, Norwegian Computing Center, Forskningsveien 1b, Oslo, Norway.

Downloaded from https://academic.oup.com/comjnl/article/15/1/32/418374 by guest on 19 April 2024