Deadlock problems of dynamic memory allocation on minicomputers with multilevel interrupt system

By J. SOMOGYI

1. Introduction

There have been a number of papers in the last decade dealing with control of concurrent processes (see [1] for a comprehensive list of papers). In this paper we investigate the applicability of some of the theoretical results to minicomputers with multilevel interrupt system.

In view of concurrent processes the main characteristics of these machines can be summarised as follows (see Fig. 1). Let i, j and k denote three interrupt levels such that $i<j<k$, and suppose that at time t_0 the machine works on level i. At time t_1 a level k interrupt request arrives. Because of $k>i$ the hardware saves the context (program counter, indicators, etc.) of level i, and loads the context of level k into the appropriate hardware registers. Then program execution goes on at the memory address pointed by the new program counter.

At time t_2 a level j interrupt request arrives. Because of $j<k$ (the current level) the request is recorded by the hardware, but not dealt with. At time t_3 level k completes. At this time level j is the highest waiting level. Therefore the hardware selects (via context changing) level j for execution. At time t_4 level j completes. Then the hardware returns to level i interrupted at time t_1.

In section 2 we shall describe a deadlock problem related to the monitor program of the VT1005, a Hungarian manufactured minicomputer with an interrupt system described before. The design and development of the monitor program was

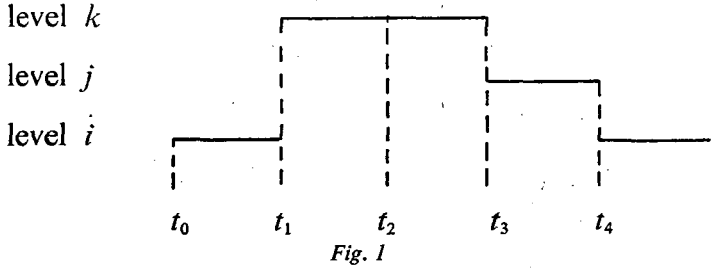

Fig. 1
Hardware scheduling of interrupt levels

performed when the complete specification of the machine had not been freezed by the manufacturer. Therefore, instead of using a simulator the monitor was written in a higher level language, CDL [2], and was debugged on another minicomputer, R10, which is equivalent to CII Mitra 15.

From programmer's point of view the interrupt system of the two minicomputers are identical, so the ascertainments of this paper apply to both machines.

In section 3 two solutions of the deadlock problem will be described and compared in view of their memory requirement. Section 4 is devoted to the implementation details. In section 5 we prove that the solution described in section 4 is deadlock free.

2. The deadlock problem

Suppose that in Fig. 1 the program executed on level i calls for a monitor service (e.g. ASCII—EBCDIC conversion), and at time t_1 the control is in the service routine. At the same time another program on level k enters, and calls for the same monitor service. Then there are two possibilities, either queuing the second and the possible further requests, or writing re-entrant service routines.

In the first case the service routines become resources, each forming a separate resource type. Moreover the service routines may call for further service routines, etc. Avoiding deadlocks so, the deadlock avoidance may become overcomplicated for the limited memory of a minicomputer.

In the second case we have to provide dynamically allocated working areas for the service routines, with memory being the only resource type to be dealt with. Further simplification can be introduced by allocating the memory in blocks of a fixed size. Though a CDL procedure is available handling variable size blocks, it does not fit 8K byte memory of our machine [3].

After all in our system there is a common memory consisting of fixed size blocks, and one CDL procedure can have one such memory block. Suppose that the programs running on interrupt levels greater than zero are all peripheral device handlers. (There is no interrupt associated with level zero, this level is reserved for user programs.) Then, in best case an interrupt level requires two memory blocks, as shown in Fig. 2.

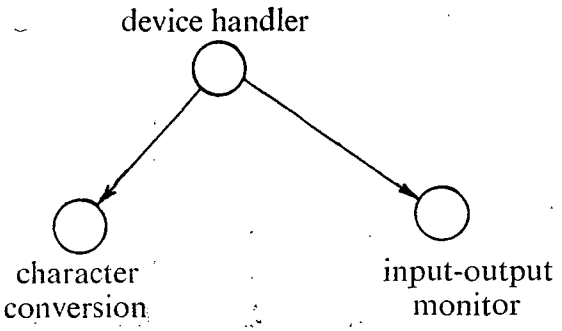

Fig. 2
The best case

In the figure CDL procedures are represented by circles. The procedures within the same row are executed one by one. Therefore the number of rows is equal to the depth of nesting, that is to the maximum number of blocks required by the interrupt level.

The worst case is caused by device errors requiring operator intervention (e.g. card jam, paper low). In this case the input-output monitor is called for sending the appropriate message to the operator's console. Then, as Fig. 3 shows, the interrupt level requires five memory blocks.

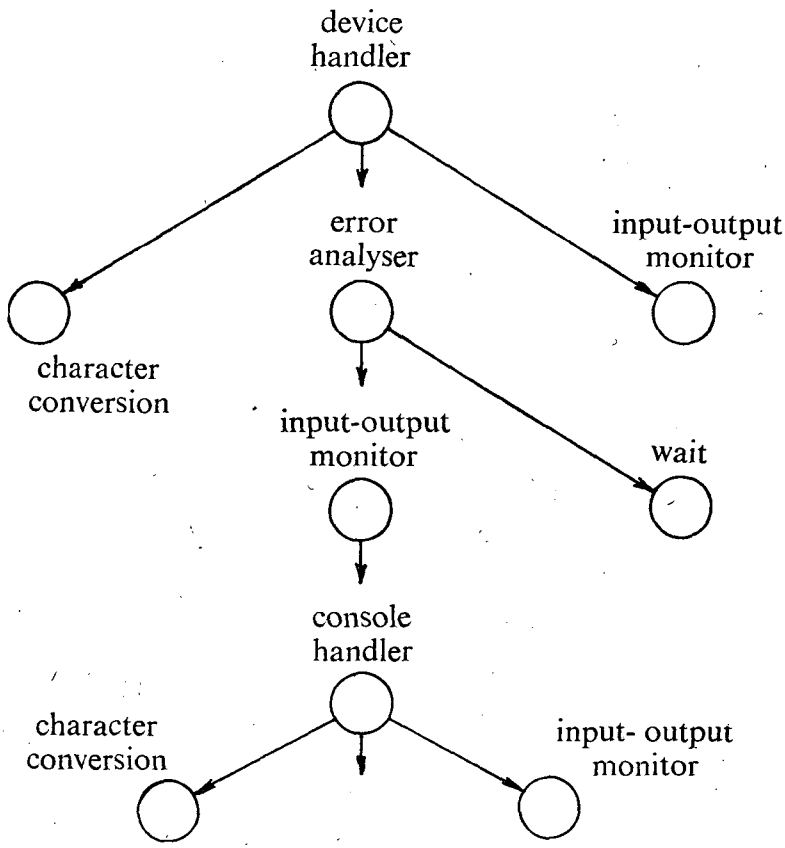

Fig. 3
The worst case

Suppose now that some of the interrupt levels have memory, but none of them has enough to complete, and the common memory has been exhausted. Then the system contains a deadlock.

3. The solution of the deadlock problem

The maximum memory requirement of each interrupt level is given. Therefore the deadlock could be prevented by the known methods [5]. Unfortunately the application of these methods to the machines described before, implies a serious efficiency problem.

When occuring an interrupt, the service must start immediately so as to avoid the loss of data or status informations. Doing so, we need at least one memory block. Therefore, if we decide to postpone the service of the interrupt for avoiding the deadlock, we must do it before starting the input-output operation, that is, the operation must not be started. However, this implies significant loss of time, because the input-output operation could take place during the waiting time.

Because of the small memory size of the VT1005, we are forced to look for a solution as simple as possible. In the following we shall compare two simple solutions in view of their memory requirement.

We are interested only in the differences of the two solutions, so we ignore the memory requirement of the user program, which is the same in the two cases.

3.1. The trivial solution. Let the size of the common memory be large enough to satisfy the memory requirement of each interrupt level simultaneously even in the worst case.

Calculating the total memory requirement we have to take into consideration, that the operator's console can service one request at a time. The other requests are queued by the input-output monitor. Therefore only one of the interrupt levels can have the maximum number of blocks, the others can require one less.

Let M denote the maximum number of blocks required by an interrupt level, let B be the length of one block, and let N be the number of interrupt levels active at a time. Then the total memory requirement is $M\ B+(N-1)(M-1)\ B$.

3.2. A nontrivial solution. Let the size of the common memory be such that the minimal memory requirement of each interrupt level could be satisfied simultaneously, and one of them could have the maximal requirement. Moreover only one of the interrupt levels at a time can have memory exceeding the minimal requirement. As a matter of fact, it would be good enough to grant one memory block per interrupt level for starting the interrupt service routine. However, we want to avoid the unnecessary suspension of levels when the input-output operation was error free.

Let M, N, B be as before, let m denote the minimum number of blocks required by an interrupt level, and let C be the size of code necessary for controlling the memory allocation according to the present solution. Then the total memory requirement is $NmB+(M-m)B+C$.

The nontrivial solution has an advantage over the trivial one, if

$$MB+(N-1)(M-1)B > NmB+(M-m)B+C$$

and, therefore, if

$$N > 1+\frac{C}{(M-m-1)B}.$$

4. The implementation

The common memory consists of $Nm+M-m$ blocks, $M-m$ of which are subject to mutual exclusion. The mutual exclusion is implemented via the enqueue and remove primitives defined below.

Let q denote the waiting queue, consisting of the total number of interrupt levels plus one element, let p be the pointer of the queue, and let n be the actual interrupt level. Then

enqueue (n):

L: $p = 0$ —— $q[0] := n$, $p := n$ —— return

$p \neq 0$ —— $q[p] := n$, $p := n$ —— DIT —— goto L

where DIT stands for Desactivate InTerrupt. This allows to continue the execution of other interrupt levels with lower priorities.

remove (n):

$p = n$ —— $p := 0$ —— return (no levels are waiting)

$p \neq n$ —— $q[0] := q[n]$ —— PIT $q[0]$ —— return

where PIT stands for Programmed InTerrupt. This activates the interrupt level desactivated by the DIT operation. The execution of the activated level will continue in the enqueue primitive just behind the DIT operation.

Dealing with a single-processor system, the primitives are implemented by interrupt inhibition.

For allocation and deallocation of memory blocks, let R denote a list consisting of the total number of interrupt levels plus one element, and let n be the level requesting or releasing a memory block.

The allocation procedure:

$R[n] := R[n]+1$ —— $R[n] \neq m+1$ —— allocate memory

$R[n] = m+1$ —— enqueue (n) —— allocate memory

The deallocation procedure:

deallocate memory —— $R[n] := R[n]-1$ —— $R[n] \neq m$ —— exit

$R[n] = m$ —— remove (n).

Returning to the question of choosing the one of the two solutions to be advantaged our numerical results are: $C=160$ bytes, $B=32$ bytes, $M=5$ and $m=2$, therefore $N \geqq 4$. Hence the second solution needs less memory, if at least four of the peripheral devices are expected to work simultaneously.

5. Proof of the nontrivial solution being deadlock free

For formal treatment of the deadlock problem we adopt the definition of [1] with the number of resource types equal to one. To begin with we convert the description of the interrupt servicing procedure into a chain. Fig. 4 shows the chain corresponding to Fig. 2.

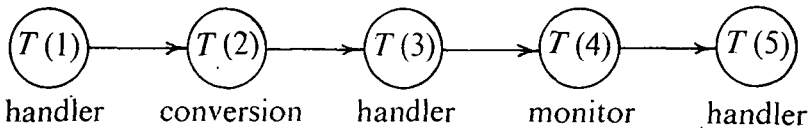

Fig. 4
Chain corresponding to Fig. 2

Within the chain $T(j)$ represents a unit of execution during which the resource usage of the chain remains constant. Such a unit will be called a task. The execution of a chain implies a sequence of task initiation and termination events. The task termination events are associated with both the releasing of resources not needed by the next task, and the immediate requesting of the additional resources necessary for initiating the next task. The task initiation events are associated with the allocation of the resources requested at the termination of the previous task.

The interrupt servicing system consists of the parallel combination of chains defined above. Then the state of the system is described by the pair of vectors

$$\underline{P}(k) = (P_1(k), \ldots, P_N(k))$$

and

$$\underline{Q}(k) = (Q_1(k), \ldots, Q_N(k))$$

where $P_i(k)$ and $Q_i(k)$ denote, respectively, the number of memory blocks held and requested by the ith chain after the kth event.

Let $\overline{T}(j)$ and $\underline{T}(j)$ denote, respectively, the initiation and termination events of task $T(j)$. Then Fig. 5 shows the P_i and Q_i values of the chain of Fig. 4.

$\overline{T}(1)$	$\underline{T}(1)$	$\overline{T}(2)$	$\underline{T}(2)$	$\overline{T}(3)$	$\underline{T}(3)$	$\overline{T}(4)$	$\underline{T}(4)$	$\overline{T}(5)$	$\underline{T}(5)$
1	1	2	1	1	1	2	1	1	0
0	1	0	0	0	1	0	0	0	0

Fig. 5
P_i and Q_i values of chain of Fig. 4

Note that Q_i can have values of 0 or 1 only. The interpretation of $Q_i(k) \neq 0$ is that the ith chain is awaiting the allocation of a memory block. $P_i(k) \neq 0$ and $Q_i(k) = 0$ implies that the ith chain is in execution.

Let w denote the system capacity, that is, the total number of memory blocks. We say that the system in the kth state contains a deadlock, if there exists a non-empty set D of chain indices such that for each i in D

$$Q_i(k) > w - \sum_{j \in D} P_j(k).$$

Applying the notation to our case, we see that $0 \leqq Q_i(k) \leqq 1$, $0 \leqq P_i(k) \leqq M$ and $w = Nm + M - m$. Suppose that there exists a subset D of chain indices such that

$$Q_i(k) > w - \sum_{j \in D} P_j(k)$$

for each $i \in D$.

We shall come to a contradiction by this. There are three cases to consider.

Case 1. $P_j(k) \leqq m$ for $j=1, 2, \ldots, N$. Then

$$w - \sum_{j \in D} P_j(k) \geqq w - |D| m \geqq w - Nm = M - m = 3 > Q_i(k)$$

for $i=1, 2, \ldots, N$.

Case 2. There is a chain index r such that

$$P_j(k) \leqq m \quad \text{if} \quad j \neq r$$

and

$$m < P_r(k) \leqq M.$$

Then we have two subcases.

Case 2A: $r \notin D$. Then

$$w - \sum_{j \in D} P_j(k) \geqq w - |D| m \geqq w - (N-1) m = M > Q_i(k)$$

for $i=1, 2, \ldots, N$.

Case 2B: $r \in D$. Then

$$w - \sum_{j \in D} P_j(k) \geqq w - P_r(k) - (|D|-1) m \geqq w - P_r(k) - (N-1) m = M - P_r(k)$$

but $P_r(k) + Q_r(k) \leqq M$, therefore $M - P_r(k) \geqq Q_r(k)$ so there is an index $r \in D$, for which

$$Q_r(k) > w - \sum_{j \in D} P_j(k)$$

does not hold.

6. Conclusions

Because of the modest instruction set of the minicomputers the size of the code increases rapidly with the complexity of the algorithm. The optimal solution can hardly be found, in general it needs lengthy experimentation. The price given for the simplicity of our algorithm is the poor utilization of the memory. We could solve the problem using less memory blocks via more complex algorithm.

The machine independently defined algorithms for deadlock avoidance could hardly decrease the timing efficiency. It is worth noting, that the machine independency versus efficiency problem did not occured in the other parts of the system, in spite that a machine independent higher level language (CDL) was used.

RESEARCH INSTITUTE FOR APPLIED
COMPUTER SCIENCE
BUDAPEST

References

[1] COFFMAN, E. G. & P. J. DENNING, *Operating systems theory*, Prentice-Hall, Englewood Cliffs, N. J., 1973.

[2] KOSTER, C. H. A., A compiler compiler, *Report* MR 127, Mathematics Centrum, Amsterdam, 1971.

[3] JACOBSON, M. & J. MÜLLER, The buddy system in CDL, Machine oriented higher level languages, North Holland Pub. Co. Amsterdam, 1974.

(Received June 8, 1976)