SETL Newsletter 98 J. Schwartz

Reflections on P. Markstein's Newsletter on SETL extensions for operating system description

Jan. 27, 1973

It is well worth trying to look from an abstract point of view at the parallel process dictions described in Newsletter 97, i.e., to ask the question: how do these notions relate to the aim, fundamental to SETL, of importing into programming the powerful descriptive primitives used in mathematics?

We can begin an answer to this question as follows. Within a purely mathematical framework, processes are ordinarily (though implicitly) assumed to be infinitely fast; moreover, one ordinarily assumes infinite amounts of 'data space' to be available, so that data objects of arbitrary size can be accommodated. If assumptions even partly approximating these are valid in a particular programming situation, the temptation to use parallel dictions will normally not be strong, since parallel algorithms are generally less transparent than sequential algorithms, and since an infinitely fast sequential algorithm will run to completion just as rapidly as any parallel algorithm. Thus one is only tempted to begin thinking about parallelism when the real fact that processes require finite time to execute enters into one's viewpoint. Taking this fact into account, and given the fact that one will normally be able to execute various types of processes in true, real time, parallelism (even though, in many cases, all but one of these processes may be constrained to be some simple I/O action), it is reasonable to begin exploring models allowing maximum amounts of parallelism to be represented. Various models of this kind have been described in the literature (by Karp, Miller, Kotov, Narinyari, and others). Generally speaking, they regard a total computation as a pattern of function applications, each of which requires the results produced one or more prior functions to begin its own course of calculation. In such a model, a function-evaluation can begin as soon as all its inputs are available; any number of evaluations can proceed in parallel; each evaluation requires some finite amount of time to go to completion.

In a real computer system computational resource will of course be limited. From this fact arises the necessity to schedule the use of available facilities by some proper subset of a family of processes, all of which simply appear in the abstract models mentioned above as 'ready to run'. Such scheduling, which in accordance with the reflections just set forth we regard as a motive necessarily implicit in any decision to make use of parallel process dictions, will typically aim to serve at least one of two related goals. On the one hand, one may schedule for efficient use of available facilities. On the other hand, one may schedule so as to guarantee timely response to paraticular external conditions. Of course, the design of any individual scheduler can reflect both these goals in mixed proportion.

A scheduler may be regarded as an ongoing, single process, aware of facts of four main kinds: of the existence of certain other processes, of properties of these processes important for scheduling, of computational facilities available for use, and finally of external data affecting scheduling priorities. Given all this information, the scheduler selects one or more processes for immediate execution and causes them to execute. This may involve suspending processes currently in execution; in general, processes can be suspended even after they are started up, though in some cases a scheduler may have to cope with processes which, once launched, cannot be halted until their natural termination.

Note in this connection that it is natural to regard the scheduler as a process which executes continually: as it were, in its own processor. Of course, even in multiprocessor configurations it will generally not be reasonable to assign a single processor full time to the task of scheduling, since a scheduler running steadily will in most cases merely reconfirm the correctness of its last previous decision Instead therefore of using a full processor for this purpose, one uses a very nuclear 'interrupt mechanism' capable, but capable only, of detecting all changes in data of interest to the scheduler, and of causing the scheduler to execute whenever such a change occurs. The basic logical situation however is exactly what it would be if the scheduler executed continually.

Having said this much, it is worth noting that a single process operating as part of a scheduled family of processes faces an environment very much like that which would confront it if true multiprocessing were in question. That is, suspendable scheduled processes face all problems of interprocess coordination which are met in a full parallel-processor situation, only qualitative details separating the one situation from the other. To see this, one has only to observe that a scheduler which caused each process known to it to execute for one cycle in rotation would create what was exactly a parallel processing environment; between this extreme case and the typical case of a scheduler executing and suspending processes in an unpredictable way there is only a quantitative difference.

A remark casting useful light on the internal structure of schedulers may be made. The urgency of reaction to situations with which a scheduler is required to deal can vary by several orders of magnitude between situations of different types. For example, to prevent over-writes a simple data-move routine operating in conjunction with a high-bandwidth external reader may have to be activated within a fraction of a millisecond after a small buffer becomes filled, whereas it may be perfectly appropriate to use tens of milliseconds in carefully choosing the next job to run on a timesharing system. This implies that particularly urgent processes will have to be scheduled more rapidly than the most complex parts of the scheduler can act. There is of course an easy way out of this superficial dilemma, namely to use not one but a layered family of schedulers. The first of these will be extremely simple and fast, and will decide whether the second level scheduler or some other process of greater urgency is to be executed; and so forth through as many layers as are necessary. The organization of such 'layering' by the use of an interrupt system is straightforward. Observe that the hardware design of interrupt systems is often such as to provide optimal primitives for use in the innermost portions of a layered scheduling system of the type envisaged.

We now return from a discussion of these questions of technique to review in more detail the goals which might lead a programmer to complicate his work by the use of scheduled-process dictions rather than generally simpler monoprocess dictions of the sort embodied in standard SETL. I list those which appear to me most plausible, as follows:

1. Maximally expeditious completion of a job, or scheduling of a sequence of jobs for maximal throughput.

2. Response, having real-time character, to external circumstances.

3. Scheduling of processes using common data, on a data availability basis.

Concerning goal (1), the following may be said. The availability of true multiprocessing configurations might lead the individual job, as opposed to the operating system, programmer to employ multiprocess scheduling within his own job as a standard technique. However, experience with such configurations is at present extremely limited, and it is dangerous to regard hypotheses concerning the probable pattern of such use as anything more than a very tentative guide to present problems of dictional design. Setting aside the consideration of multiprocessor hardware configurations, we come to the judgement that the single-job programmer will normally not wish to treat his job as a set of potentially parallel processes to be scheduled by a private scheduler which he supplies. The complications of this approach will in most cases outweigh its possible advantages, though it is possible to imagine a single-job programmer aiming at an I/O buffer-management scheme complicated enough to make a fair degree of internal subprocess scheduling attractive. Nevertheless, it seems probable that only processes which collect and coordinate tasks arising independently and externally, i.e., only processes having in substantial degree the character of an operating system, will find the use of parallel-process dictions easy enough, and the gains from their use substantial enough, for such dictions to find more than occasional use. However, the programmer of processes whose main purpose is the coordination of other independent processes will in some cases prefer to be able to receive software interrupts even from devices over which he does not have direct physical control. For example, the scheduler routine for a simple data retrieval system, even one that uses the services of a quite autonomous operating system for actual file access, might have to be executed each time a read operation was complete, in order to determine which of a number of independent processes all requesting access to a given file element was to be executed. These last considerations emphasize the fact that it can be valuable, in putting together the base-level interpreter which defines a semantic structure, to establish a hierarchical family of interrupt conventions which allows any process in a total family of processes receive an interrupt.

The preceding discussion confirms that goals (1) and (2) above, rather than goal (3) are likely to lead a programmer to use scheduled-process dictions systematically. Goal (1) typifies operating systems; goal (2) typifies real-time control systems. It must now be noted that the operating system designer faces a fundamental problem which does not trouble the designer of real-time control systems. The processes executed in a real time control system can be assumed to cooperate harmoniously. That is, an operating system must allow undebugged processes, i.e., processes which will certainly attempt to perform entirely perverse actions, to execute at least temporarily. Thus an operating system design must address an entire range of protection problems; problems which can be avoided in a real-time system design. For this reason real-time and specialized data-availability driven systems are essentially simpler than full operating systems. Before going on to survey the charcteristic problems of protection, we therefore choose to round out our discussion of scheduled-process dictions in an unprotected environment by saying something more concerning the use of an extended SETL for the description of these simpler systems.

In providing such a system of dictions, one's essential aim is to define a logical framework conducing toward a highly modular description of general scheduling processes; at the same time, these dictions should not *preclude* (though they need not *imply*) ultimately efficient implementation. For systems of the type considered, much of what is necessary can be obtained by using scheduling system built around a set of priority queues. The scheme which this consideration suggests may be sketched as follows. Operations which are not performed 'in line' as part of an ongoing single process can be posted, with a stated priority n, on the work-queue of a suitable 'facility'. With each facility there will be associated a process P which 'serves' the facility, i.e., which attends to the work deposited on the facility's work queue. Moreover, with each facility F there will also be associated one or more processes which schedule the facility. An appropriate scheduler S will either be executed or enqueued for execution (on the CPU facility) whenever work is enqueued on the facility F for which S schedules (within some range $\min \le n \le \max$ of priorities). The facility service-process P will operate in a relatively automatic way, merely dispatching the first item from its highest-priority nonempty work queue, monitoring the progress of operations, and transmitting appropriate 'operation-complete' messages, perhaps with substantial associated data blocks, to the process which initiated the request for an enqueued service. Note that primitives must be provided which allow data-blocks to be transferred back and forth between a process enqueueing a service request and the server which executes this request. A reasonable communication convention is to have the server insert such a data block as the k-th component of a *communication vector* associated with each process in a standard way; for example, the communication vector may itself be some standard component of the state-vector defining a process.

The scheduling process S operating in conjunction with a particular facility F, and for work enqueued in a given range of priority on the use of this facility, will itself have a definite execution priority m. The enqueueing macro which enqueues work to be scheduled by S should have essentially the following form:

```
if m gt currentpriority then
     savepriority = currentpriority; currentpriority=m;
     enter service request into appropriate queue;
     place current process on 'request CPU' queue, with
      priority value =  savepriority;
     transfer to execute scheduler of priority level m;
else enter service request onto appropriate queue and
 continue executing.
```

Note that this enqueuing form deviates somewhat from that proposed by Markstein, in that priorities enter in an explicit way.

The relatively straightforward system of inter-process linkages outlined above should allow a wide range of real-time control and data-availability driven systems to be described in a transparent, modular way. Moreover, they should encourage the development of orderly, efficient work-flow patterns. However, in designing particularly complex systems, it might be desirable to allow an additional level of modularization, namely to allow secondary systems of work queues to be maintained for secondary task groups, with a secondary family of priority-organized schedulers examining each secondary system of workqueues and transferring items from these queues to central queues for execution.

Note in connection with the above that it is possible and desirable to allow all schedulers and facility service programs to operate with all levels of interrupt enabled (except for short periods of time). Moreover, these processes need and should not reserve exclusive access to the work queues with which they are concerned. (This is an important design consideration: It would be quite undesirable, for example, to have a low priority scheduler prevent a higher priority scheduler from accessing some workqueue of interest to both these processes.) The following technique can be used to avoid undesirable acts of data-structure reservation: a scheduler or service process P can, as a stnadard matter, execute an await C whenever it believes that no more work remains for it; here C designates the condition required for P to begin its next cycle of execution. If work to be performed by P has been posted by a higher-priority process even before this await is executed, P will begin a new execution cycle immediately; if not, it will actually be suspended until some other process' activity causes C to be satisfied.

Note that the modular process-coordination techniques which have been suggested make heavy use of the await diction. It will be particularly common for processes to be suspended awaiting some change in their own communication vector. All in all, the optimization of await requests emerges as an issue important for the efficiency of systems using the design approach suggested above. Various manual techniques potentially useful for such optimization suggest themselves. For example, conditions awaited can be classified into coarse categories and re-evaluated only when coarse *a priori* evidence indicates that they might possibly be satisfied. Certain manual techniques for the optimization of await requests might lend themselves to automatic implementation. This whole question is deserving of further study.

The techniques suggested above should suffice to handle most situations in which service requests can be posted to some single facility for disposition. More severe problems will arise in connection with services which can only be supplied by the coordinated use of several independent facilities. These are the situations in which specially programmed, carefully thought out schedulers are apt to be required. It may be anticipated that a coordination problem will arise most commonly in securing the central memory space necessary for a secondary memory or I/O transfer operation to be initiated. However, other cases of coordinated facility use will undoubtedly be encountered; careful consideration of representative examples is necessary if optimal approaches to these coordination problems are to be elucidated.

An important technical problem must be solved if interrupts are to be handled within a SETL or even BALM-like semantic framework providing a garbage-collected memory millieu. Namely, the garbage collector must be made interruptable; moreover, high priority processes must be able to secure memory even after the garbage collector has been set into motion by a process of lower priority. A scheme