Basic supervisor facilities for real-time

I.C. PYLE
AERE Harwell, England

Present address:
Department of Computation, University of York, England

An analysis of the supervisor facilities in a number of real-time operating systems leads to the specification of a number of basic facilities. These cover task handling, resource allocation, and inter-process communication. The parameters required for these facilities include identification of tasks and identification of peripheral devices. The handling of I/O may be treated as an autonomous process to carry out the transput, with inter-process communication transferring the data to the main process.

Introduction

Several papers at the First European Seminar on Real-Time Programming discussed the interface between the real-time programming language and the real-time operating system. In our investigation of CORAL 66 as a programming language for real-time work, on a PDP-11 with the Disk Operating System (DOS), we have been seeking to determine a suitable set of basic supervisor facilities which we could implement by elaboration of the DOS Monitor. Our hope is to establish a standard interface between a running CORAL 66 program and the supervisor (or run-time operating system) controlling it.

CORAL 66 as officially defined (Woodward, Weatherall and Gorman, 1970) does not require a run-time operating system. In practice, programs written in CORAL normally will use a run-time operating system (i.e. a supervisor program) so that the program as written does not have to be concerned with the details of peripheral handling, and to permit multiprogramming. This is an extension of the concept of a library, which is in official CORAL 66. The other point about the official non-insistence on a supervisor is that it enables the supervisor itself to be written as a CORAL 66 program. This has been done for the Myriad Computer, making the supervisor called MOLCAP (Jackson, 1970).

An attempt to standardise supervisor calls amounts to an extension of the CORAL 66 definition, by establishing certain facilities which would then be common to all CORALS at the appropriate level. In order to make the definition strong enough, we have to define the facilities to be provided by the supervisor with great care and introduce a suitable notation by which they may be specified in a CORAL program. The latter part, the syntax, is not difficult. The major part of the problem is in specifying the facilities.

Supervisor facilities

There are three principal categories of supervisor facility: Task (or Process) facilities; Resource Allocation facilities; Communication facilities. The first of these is actually a special resource allocation (since it is concerned with the allocation of the processor in the computer), but it is sufficiently distinctive to warrant a separate category. The third covers what is normally called I/O, and also inter-process communication. I/O devices may be simple slaves to the main processor, requiring simple communication facilities, or they may have autonomous capabilities in which case they are best regarded as separate processors running in parallel with the control or computing processor(s), and the communication between the processes propelled by these processors is inter-process communication.

The principle of selection of facilities for a prospective standard must be to keep the list short and simple. The supervisor occupies valuable store during run time, and must not be made unnecessarily elaborate. If the standard covers more than the bare minimum the supervisor program will provide the facilities whether or not the application program requires them. Any enhancements needed to elaborate the supervisor facilities can be included in library procedures to be loaded only when required, which call on the supervisor for the primitive facilities. Accordingly, we proceed by giving a list of facilities which appear to be primitive.

The entry conventions for supervisor calls will have to be settled for each particular implementation, but the standard must specify what parameters are to be transferred for each supervisor call. Some computers will have an established supervisor call instruction, others will use a trap or entry to a standard location, or force

some special exception to occur. The standard cannot expect to lay down which method shall be used, but can reasonably assume that there will be just one general method of supervisor call in each implementation, which is different from an ordinary procedure call (for example, to cause a change of processor state where there is a difference between user mode and supervisor mode).

Task facilities

The facility which almost all supervisors will provide is 'generate task'. This will activate an already loaded program to operate on given data in the store, at a specified execution priority; the supervisor will provide the identification for the task. Thus there are four parameters, three input and one output:

entry location	(a destination)
data origin	(a pointer)
priority	(an integer)
task identification	(output, probably a pointer).

The priority may be omitted: the default value will be the priority of the task issuing the supervisor call. There may be rules prohibiting a task from generating other tasks of higher priority than itself. Improved ideas on task scheduling may lead to a better method of regulating the allocation of processors to processes, but at present the use of a priority number is the most suitable.

The next and most fundamental supervisor facility is 'terminate task'. Every program needs to be able to specify its dynamic end. With computers which have a 'stop' instruction, the first rule for programmers is not to use it, especially in real-time work. There can be two variants of this supervisor call: terminate own task (without parameter) and terminate other task (with other task identification as parameter). This introduces the need for task identification and the problem of naming tasks. The standard cannot solve the problem, but must assume that a solution is chosen in a particular implementation, and any restrictions consequent on that solution limit the range of other tasks which can be controlled. The two variants can be viewed as the same supervisor call, with a default value for the parameter if it is omitted, being 'self'. For a discussion of the problems of task identification, see Taylor (1972).

As an alternative method of terminating a task, but denoting an abnormal or exceptional dynamic end, it is desirable to have a separate supervisor call: 'abnormal termination'. This applies only to the current task, and carries a parameter which can express the nature of the abnormality or exception.

A task which may be subject to abnormal termination from another task should be given the ability to specify its own local termination, to do any tidying up necessary. The local termination would be a procedure, which is activated by the supervisor immediately after a supervisor call to terminate. It would have one input parameter, being the termination code. There has to be a supervisor call to pass this local termination procedure to the supervisor for the task. If there has been no such procedure specified before termination, then the supervisor removes the task immediately after a supervisor call to terminate; otherwise it activates the local termination procedure (after removing the record of it from the task entry), and on return from it removes the task. Thus we have a supervisor call

set task termination procedure (p)

with one parameter (of type PROCEDURE (INTEGER VALUE).

A task in a real-time system may require a delay of a period of time. We therefore need

wait (number of time intervals)

where the magnitude of the time interval will be system-dependent.

Delayed activation

A powerful technique, especially useful in real-time for dealing with time-out situations, is a delayed activation which can be cancelled. With an activate which takes effect after a specified time delay, the time-out action can be easily set up, leaving the supervisor to look after the timing. The additional facility which is necessary is to be able to cancel the delayed activation, if the awaited response arrives in time. Since this would be the normal situation, the supervisor must keep its delayed activation requests in a way which permits efficient cancellation.

This can be treated as a combination of the supervisor calls to activate and wait (activate the time-out action immediately, but put an appropriate wait at the beginning of the task). The cancel is then simply a call to destroy the task, which will take place while it is in the wait state. There is a danger of trouble if there are other waits in the time-out action task, since the task might then get destroyed in a partially completed state. This can be solved by the discipline of insisting that such a time-out action contains no further waits, although it may in turn activate a further task (which would not get destroyed when the response comes).

Task synchronisation and interrupts

The well known primitives for handling semaphors permit a task to be held up at defined points until another task gives it the go-ahead:

Secure (semaphore in*)
Release (semaphore in*)

There are arguments in favour of a further supervisor call, which cannot be implemented by use of the above:

attempt to secure (semaphore in*, status out*)

This is equivalent to secure if the semaphore would permit the task to proceed, but otherwise gives the status of the semaphore without suspending the task. Releasing a semaphore gives a stimulus to the task which had previously made a secure for it.

Stimuli can arise from outside the computer (hardware interrupts) or from other running tasks. The effect of the stimulus is not directly to create a task, but to resume execution of a task which must previously have been set up and suspended awaiting the stimulus. In order to deal with critical sections, it is necessary to be able to inhibit the response to stimuli for certain periods of time.

Thus the primitives required are:

Set response to stimulus (stimulus identity in, semaphore out)
Inhibit response to stimulus (stimulus identity in)
Permit response to stimulus (stimulus identity in)

In the task to be performed, it is normally held up by the semaphore; it starts in response to the stimulus, and must return to the semaphore wait again if it is prepared to respond to further stimuli.

The principal problem about this technique is identifying the stimuli between separate processes. In a simple system, the natural solution is to adopt a system-wide assignment of stimulus type numbers corresponding to distinct physical interrupts at the hardware level.

Resource allocation

The allocation of resources other than processors and main store will assume that they are discrete, and have in general to be allocated in two stages: shared and exclusive. Every resource has initially to be reserved, and it will in principle be shared. This means that there need be no time delays on reservation. A claimed resource can then be secured for exclusive access, and must be released when exclusive access is no longer needed. The resource must be discarded when it is no longer needed. A subsequent reservation will not necessarily get the same device, assuming that there are several of the same type forming the resource. Depending on the nature of the resource, different things may be done when the resource is claimed and secured.

Thus we need two supervisor calls:

Reserve resource (resource type in, identity out)
Discard resource (identity in)

To handle the exclusive use of a resource, we need to use a semaphore which will control access to that resource:

Prepare for multiple use (identity in, semaphore out)

The exclusive uses are then controlled by the use of the semaphore.

Communication

We assume that the executing task may communicate with a number of data sets outside itself, which may be accessed sequentially. Streams of data will be either input or output, and may be buffered. A stream which is output from one task may be input to another task, or after it has been closed, to the same task. Buffering implies having separate autonomous tasks (usually very simple) complementary to the main task: if the main task puts information to an output stream for printing, then an auxiliary task has to read the output buffer and print what it finds there.

We have to consider input output for different categories of devices, depending on the nature of the information they transmit in an elementary operation. Basically, there are character devices (e.g. teletype), word devices (e.g. ADC samplers) and block devices (e.g. disk). For character devices, it is usual to deal with streams of characters, and it is more economical to specify the source or sink separately, rather than with each individual input or output character

Set source (device identity in)
Set sink (device identity in)
Input character (character out)
Output character (character in)

For the character devices, it is often desirable to be able to translate the character code, so that a uniform internal character code can be used, but this is more economically done by a library procedure on a complete line of characters together. Words are input and output from explicitly identified devices:

* The suffixes 'in' and 'out' distinguish input and output parameters of the supervisor call.

Input word (device identity in, word in)
Output word (device identity in, word out)

For block devices, a buffer area is needed, which has to be set up by the program in a special format, and then given as a parameter in the supervisor call. This can include the specification of which area on the device is to be used, and the direction of the transfer:

Put block (buffer in)

Storage allocation

No run-time supervisor facilities are proposed for allocation of main storage. This is a deliberate restriction on the flexibility of the operational system, which corresponds to the essential characteristics of the situation, at least at the present state of the art. The argument for excluding them rests on the requirement that the real-time programming language be predictable.

If there is run-time allocation of storage by the supervisor, then a general strategy has to be adopted to deal with block of various sizes and unco-ordinated requests and releases. Consequently store fragmentation is a real danger, and from time to time an unpredictable store jam will arise; in this event, either the system will be brought to a complete halt or there will be a significant time delay while there is some reassignment of storage.

The implications of this decision are that the allocation of storage is done at load time, with the user's program thereafter responsible for how the store is used. Thus, for example, the program can state that it needs a stack, and must specify to the loader an appropriate maximum size for this stack. Space will be allocated permanently for the maximum size stack, and within this the user program keeps its stacked variables and its own stack pointer. Separately, the program can state that it needs some buffers of a certain size, and must specify to the loader an appropriate maximum number of buffers which can even be concurrently in use. Space will be allocated permanently for the maximum number of buffers, and within this it is up to the user program to control its own use of the buffers, and take its own action if the situation arises when all buffers are in use and it needs another one.

In other words, the organisation of store provided is fixed at load time as far as the supervisor is concerned, and any dynamic usage is directly controlled by the user program. This is in accord with the views expressed at the First European Seminar on Real Time Programming (pages 31 and 32, speakers D and G).

Conclusion

This paper has begun the process of specifying the supervisor interface, by concentrating on the areas of tasks, communication and resources. The interface does not specify any facilities referring to protection between processes or store areas, because it assumes that this matter is handled entirely within the supervisor, and needs no further information or stimuli from the work program. It has not covered store allocation or file structures, so is far from complete for a full operating system. Likewise it does not include backing store usage or swapping facilities. Nevertheless, facilities such as these will be needed within some real-time operating systems. At the First European Seminar on Real Time Programming, it was thought premature to make such an attempt. It is still premature to expect a standard to be widely adopted, but it is not too early to start thinking about the problem.

1. JACKSON, K., 'Myriad Library Part 2.1', RRE (1970).
2. TAYLOR, J.R., 'Control of names in systems with dynamically created objects', AERE-R (1972).
3. WOODWARD, P.M., WEATHERALL, P.R., and GORMON, B., 'Official Definition of CORAL 66', HMSO (1970).

Discussion

Q. Why do you have no absolute delay?

A. The wait specified is absolute. There is no relative delay because it can be derived from an absolute delay.

C. In my applications, delay timing has not been critical. Swapping can cause timing problems, though.

Q. How do you deal with random access devices, e.g. disk or other backing store?

A. Not in the level of supervisor I have described, but at a higher level. (Similarly for CAMAC.)

Q. Does this also apply to shared devices such as shared typewriters?

A. Yes: the (reentrable) program to handle them would be written to use the supervisor facilities described.

C. I have found that this tends to cause rather high overheads. For this reason I would include the facilities in the basic supervisor, although conceptually they belong at a higher level.

Q. How do you have reentrable programs without dynamic storage?

A. Each task must take care of its own storage allocation.

Q. How do you wait for events?

A. By semaphores.