Response to A Note on Structured Interrupts

Ted Hills
226 Shunpike Rd.
Chatham, NJ 07928
hillst@iia.org

Abstract

In **Structured Interrupts** [2] I propose a machine-independent mechanism for handling hardware and software interrupts. In **A Note on Structured Interrupts** [1], Pradeep Hatkanagalekar points out several inadequacies in the model I proposed in **Structured Interrupts**, and suggests solutions. I concur with most of his suggestions, but differ with the last.

Semantics of wait

Mr. Hatkanagalekar supplies a corrected version of the semantics of my `wait` subroutine:

> *The subroutine* `wait` *operates in an un-interruptible mode. It first examines if any of the signals for which the process has registered itself are set and passes control to the corresponding label if one is found. Otherwise it takes the current process off the CPU, and starts the now highest priority process.*

Busy Wait

The addition of a non-waiting `check_signals` is a welcome addition to the Structured Interrupts mechanism, for use in multi-processor applications:

> *We propose an explicit subroutine* `check_signals` *to be called without any parameters that would check if any signals for which the process has registered itself are set and would pass control to the label corresponding to the one which is set. The call would execute a normal return if none are set.*

Non-Deterministic Real-Time Response

Mr. Hatkanagalekar explains the following scenario.

> A process P awaits two events E_1 and E_2 where the priority of E_1 is less than that of E_2. Therefore, as per the guideline in [2], the priority of P is

> equal to that of the event E_2. After P issues a `wait`, let E_1 occur. Thus P would start running at the label associated with E_1. Now before responding to the event, P would drop its priority to that corresponding to E_1. Now let E_2 occur. This interrupt would be accepted by the CPU since its current priority permits it. However, currently, the process is in a running state handling the event E_1. Therefore the handling of E_2 would take place only when, at the end of handling of E_1, P issues the `wait` call again. Thus we have a situation where the response to the higher priority event E_2 has been delayed by the processing of a lower priority event E_1.

The "quick fix" to the mechanism is proposed as:

> *It is further required that a process may await at the most one external (hardware) event.*

The problem identified is actually worse than Mr. Hatkanagalekar indicates, but, fortunately, there is a more general solution than the one proposed.

The problem identified is that the processing of a lower priority event can delay the processing of a higher priority event. Whether this problem is acceptable in a given situation depends on the application in question, but let us assume that it is unacceptable. This problem is directly caused by the design of a single process to handle two events with different priorities. The Structured Interrupts mechanism contributes to the problem by allowing such a design.

The more severe problem is that, in this scenario, the CPU accepts an interrupt request on behalf of a process which is already in the Ready state. This is a violation of a basic design element, that a CPU only accept an interrupt request if it is known to cause the readying of a process awaiting the associated signal. The CPU would have to accept and then ignore the interrupt request. No information would be lost, since when process P issues its second wait call, it sees that event E_2 has occured, and transfers control to the associated label. However, violating this basic principle reduces the efficiency of the mechanism.

The solution is to *require that a process always run at a priority equal to or greater than the highest priority of any of the external (hardware) events it is awaiting.* This retains the flexibility of the mechanism which allows multiple events to be awaited by a single process. If a process wishes to lower its priority, it is first required to stop awaiting external signals (by using the `ignore_signal` call) which have priority greater than the process's new, lower priority. This preserves the guarantee that a CPU will only accept an interrupt request if it is known to cause the readying of a process awaiting the associated signal.

Let us revisit Mr. Hatkanagalekar's scenario with this change in mind. Event E_1 has occurred, and process P is now running with priority equal to that of event E_2, which is higher than E_1's priority. Process P has two choices. It may `ignore_signal`(E_2) and then lower its priority to that of E_1, or it may continue responding to event E_1 with the higher priority of E_2. In

either case, the CPU will not accept an interrupt request as a result of event E_2 occurring. This preserves the semantics of the Structured Interrupt mechanism. It also preserves the problem pointed out by Mr. Hatkanagalekar, that the processing of event E_2 is delayed by the lower priority processing of event E_1.

The correct solution to this problem is to have two processes, P_1 with the priority of E_1 to service event E_1, and P_2 with the priority of E_2 to service E_2. Suppose event E_1 has occurred, and process P_1 is the Running process. When event E_2 occurs, the CPU will accept the resultant interrupt request, since the priority of E_2 is greater than the priority of P_1. Process P_2 will be made Ready, and will preempt process P_1, becoming the Running process.

Summary

A Note on Structured Interrupts corrects an error in the Structured Interrupts mechanism as originally described, makes a valuable addition to its semantics for multi-processing, and highlights a problem with the priorities of processes awaiting external events, which is sufficiently solved herein.

References

[1] Hatkanagalekar, Pradeep. A Note on Structured Interrupts. Operating Systems Review, 28(2), pp. 88-91.

[2] Hills, Ted. Structured Interrupts. Operating Systems Review, 27(1), pp. 51-68.