A Note on **Structured Interrupts**

Pradeep Hatkanagalekar
Dept. of Computer Science & Engineering
IIT, Bombay, India
hattu@cse.iitb.ernet.in

January 31, 1994

Abstract

In "Structured Interrupts" Ted Hills[3] proposed a framework to model interrupts so that the traditional *machine dependent* Interrupt-Handling can be analyzed in a *machine independent* fashion. In this short paper we point out certain faults/inadequacies in the proposed model and suggest modifications to it to take care of the same.

1 Introduction

In his paper on "Structured Interrupts"[3], Ted Hills discusses how confusion exists in clear understanding of the notion of interrupts and interrupt-handling by citing the definitions of the concept in various reference manuals of some popular computers. Then he proposes a clear definition of interrupts, a universal interrupt-handler and then gives an interface model for processes to await interrupts. Later on, he makes further suggestions to use this framework to tackle real-time issues.

In this paper we point out one flaw in his model which could lead to hanging interrupts and then indicate two more deficiencies of the model along with suggestions for corrective modifications to the model.

In the following three sections, we first present the relevant part of the proposed model, describe the flaw/deficiency and then propose our modifications.

2 Uncaught Interrupts

Given below is the summary of the relevant part of Hills' model for structured interrupt-handling proposed in [3]:

1. A **signal** is a single bit, the setting of which is an **event** and can cause an interrupt. An **interrupt** is the change in CPU control flow in response to an **external event**, in order to ready processes awaiting the setting of the corresponding signal.
2. A process registers itself for a certain signal by making the **wait_on_signal** call. This takes two parameters: the **signal_id** indicating the signal of interest and a **label** giving an alternative return address. The label must be from the same stack-frame context as the one from which the call is being made. A process may make multiple **wait_on_signal** calls to register itself for multiple signals and may do any amount of processing between two calls. The call examines *all* the signals (including the current one) for which the process has registered itself. It returns control to the label corresponding to the signal if the interrupt-event has occurred, else it does a normal return.

3. After registering itself for one or more signals, a process calls **wait**. "This subroutine makes the calling process not ready, taking the process off the ready queue and allowing other processes to become the running process."
4. A universal interrupt handler operates in a un-interruptible mode, puts the processes awaiting the signal corresponding to the interrupt on the ready-queue and gives control to the current highest priority process. If no process is waiting, the event is ignored. Specifically, "a signal which becomes set cannot affect the control flow while that process is running normally."

The flaw here is that if an interrupt is delivered *between* the last call to **wait_on_signal** by a process and the call to **wait**, the signal will remain un-delivered to the process. To take care of this problem we propose the following enhanced semantics of the **wait** processing:

> *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.*

Note 2.1: On page 59, under the section titled "Usage", the paper mentions that, "A signal which becomes set cannot affect the control flow within a process while it is running normally; it may only affect control flow while the process is issuing a **wait_on_signal** call or after it has issued a **wait** call". However the indirectly implied semantics that a wait call may affect the control flow in the event that some signal of interest is already set, is not captured in the description/definition of the wait call on page 58 nor does the above statement bring out the need to do so in an un-interruptible mode.

Note 2.2: The flaw mentioned in this section becomes irrelevant when Hills extends his model for Real-Time refinements explained in Section 4 of this paper. However; he clearly intends to project the earlier part of the model as complete though not suitable for Real-Time considerations ... "Without further refinements, this scheme would be inefficient and could render a computer system useless for handling real-time events".

3 Need for Busy-Wait

After proposing the model for the handling of hardware generated interrupts - termed as *external events*, Hills proposes the interface function **set_signal** with signal_id as a parameter to widen the effectiveness of the model for the handling of Software Interrupts. This subroutine sets the bit corresponding to the signal_id and then mimics the functionality of the universal interrupt handler whereby processes awaiting the software interrupt would be readied.

In the case of Multiprocessor systems, it is a well established practice that if the *expected wait time* for some resource (the software interrupt in this case) is less than the time required for a context switch and back, it is more CPU-efficient to spin on the resource rather than release the processor[2].

In Hills' model there is no provision for *busy-wait*ing on a signal since after making the required number of **wait_on_signal** calls a process is expected to call **wait** which would incur the cost of context switches.

> *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.*

Note 3.1: Though its effect is not clearly stated, to achieve the effect of busy wait, a process could keep making the **wait_on_signal** call in a tight loop for any one of the signal_ids of interest. However; an explicit interface call to cater to this requirement is preferred.

4 Non-Deterministic Real-Time Response

To enhance the model for Real-Time characteristics, Hills proposes the following architectural changes and guidelines for usage:

1. Zero or one processes may wait at any given time for a signal associated with an external hardware event.
2. Processes have priorities and the priority associated with the CPU is the priority of the currently running process.
3. Each interrupt or external hardware event is associated with a priority value. An interrupt request is accepted by the CPU only if a process is waiting for the associated event and the (hardware) priority of the event is greater than that of the running process.
4. The priority of a process waiting for an external event must be greater than or equal to the (hardware) priority of that event.

In the section titled "More Usage", the paper says ... "In the situation that a process wishes to wait on multiple events of varying degree of urgency, the process may set its priority equal to the highest of the priorities of the concerned events before issuing the series of **wait_on_signal** calls. After receiving control at some point because a particular signal is set, the process would set its priority equal to that of the corresponding event and then respond to the event".

This can lead to the following undesirable situation: 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 the above paragraph, 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 the 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.

Remaining within the framework of Hills' model, we suggest the following quick solution to tackle this problem:

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

Note 4.1: One may wish to extend this to other types of (software) events ...In a Multiprocessor environment while P above is handling E_1, the other events may only be caused by processes running on other processors. Thus, from the point of view of this model, the event could be treated on par with other external events.

Note 4.2: We have suggested a solution that naturally fits in the mode l suggested by the paper. However a different solution to tackle this problem could be envisaged: In the situation where a process is required to preempt itself, the current processor state along with the priority could be saved on the stack without changing the stack-frame context and control given to the new label. At the end of processing when the process executes the **wait** call, the old processor state could be loaded back from the stack and the interrupted low priority processing could be resumed. In this case, rather than waking up the process with the same priority as the one with which it made the **wait** call, we suggest that the *priority on wakeup* could be introduced as an additional argument to the **wait_on_signal** call similar to the internal sleep call of the UNIX [1]. This would obviate the need for adjusting the priority in the handling of the received interrupt signal.

5 Conclusions

In this paper we have identified certain lacunae in the interrupt model proposed by Hills[3] which could lead to lost interrupts or non-deterministic delays in real-time processing; and we have suggested modifications/solutions to take care of them.

References

[1] Maurice J Bach. The Design of the UNIX Operating System. *Prentice-Hall, Inc.*, New Jersey, 1986.

[2] Anna Karlin, Kai Li, Mark S Manasse, Susan Owicki. Empirical Studies of competitive spinning for a shared-memory multiprocessor. *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pp. 41-55, Oct 1991.

[3] Ted Hills; *Structured Interrupts*; O S Reviews; V27, N1, Jan 93; p51-68.