Structured Interrupts

Ted Hills
Dynamic Digital Technology
226 Shunpike Rd.
Chatham, NJ 07928

Abstract

In this paper I refine the concept of *interrupt* to define a more structured mechanism with straightforward semantics. Some commonly implemented interrupt mechanisms are reviewed; their semantics are deduced and analyzed. I show how a different understanding of interrupts resolves the deficiencies of existing interrupt mechanisms and provides the programmer with powerful tools for addressing real-time and multiprocessing problems. On this basis a recommendation is made for architectural and operating system support of the improved interrupt mechanism.

Introduction

A goal of computer science research is the reduction of various programming problems to well-defined programming constructs with solid theoretical foundations, in order to simplify the development of correct programs. Even though the concept of *interrupts* has been with us for quite some time, I believe that the theoretical understanding of interrupts is still weak. There are clues to this weakness. For instance, in most operating systems it is necessary to construct interrupt handlers outside the normal process structure of a computer system. Another indication is that almost every implementation of interrupt mechanisms on the popular von Neumann processor architectures differs in some significant way from all the others.

If we can get a better grasp of what an interrupt is, and how to handle it, it should become easier to incorporate interrupt handling not only in device drivers well buried in operating systems, but also in application programs running at a protected level of an operating system. It could become trivial to write a program that can wait simultaneously for various hardware and non-hardware events, all at an application level and in a high-level language. This could bring real-time problems a little closer to the realm of everyday programming.

Much of this paper will be devoted to arriving at precise definitions for terms commonly used, often colloquially, within the realm of computer science. To begin with, we will examine some popular computer architectures to help us deduce a definition for the term **interrupt**.

Defining Interrupts

What is an interrupt?

The terms *interrupt*, *exception*, *trap*, and *supervisor call* are all wrapped up together when discussing computer architecture. In connection with these are the terms *event*, *signal*, and *condition*. In order to understand the essence of what an interrupt is, we will look at how the term is used by various computer manufacturers. Below I quote definitions given by the reference manuals of some popular computers.

S/370: "The interruption system permits the CPU to change its state as a result of conditions external to the system, within the system, or within the CPU itself. To permit fast response to conditions of high priority and immediate recognition of the type of condition, interruption conditions are grouped into six classes: input-output, external, program, supervisor call, machine check, and restart."[1]

NS32000: "Program *exceptions* are conditions which alter the normal sequence of instruction execution, causing the processor to suspend the current process and call the operating system for service. An exception resulting from the activity of a source external to the processor is known as an *interrupt*; an exception which is initiated by some action or condition in the program itself is called a *trap*. Thus, an interrupt need have no relationship to the executing program, while a trap is caused by the executing program and will recur each time the program is executed."[2]

MC68020: "Exceptions can be generated by either internal or external causes. The externally generated exceptions are the interrupts, the bus error, and reset requests. The interrupts are requests from peripheral devices for processor action while the bus error and reset pins are used for access control and processor restart. The internally generated exceptions come from instructions, address errors, tracing, or breakpoints."[3]

i80386: "Interrupts and exceptions are special kinds of control transfer; they work somewhat like unprogrammed CALLs. They alter the normal program flow to handle external events or to report errors or exceptional conditions. The difference between interrupts and exceptions is that interrupts are used to handle asynchronous events external to the processor, but exceptions handle conditions detected by the processor itself in the course of executing instructions."[4]

SPARC: "A **trap** is a vectored transfer of control to the operating system through a special trap table that contains the first 4 instructions of each trap handler. . . . A trap may be caused by an instruction-induced **exception**, or by an external **interrupt request** not directly related to a particular instruction."[5]

Look at the variety of terminology! In the NS32000 and MC68000 architectures, an interrupt is a subtype of exception, while in the i80386 architecture, interrupts and exceptions are both special types of control transfer. In the SPARC architecture, interrupt requests and exceptions cause traps. The NS32000 reference says that an interrupt, being an exception, is a condition. The S/370 reference avoids using the term *interrupt*, but includes all sorts of things under "interruption condition". Except for the S/370 reference, we have the common idea that interrupts, whether condition, trap, or control transfer, *have their cause external and asynchronous to the program being executed.* For now we will concentrate our discussion on interrupts which meet these criteria (external and asynchronous causes). We will later examine other types of so-called interrupts.

These disparate definitions still have enough common concepts that I think we can formulate a working definition for the term *interrupt* which accurately reflects what all these manufacturers are trying to capture in their implementations. Let me propose that an **interrupt** is *a change in CPU instruction sequence which is not caused or specified by the instruction sequence being executed at the time of the interrupt but is caused by an interrupting event external to that instruction sequence.* The event "interrupts" the programmed control flow. This definition has been carefully worded to eliminate from discussion certain things which we may now call "pseudo-interrupts", and which will be described below.

Let me emphasize the distinctions in the terms I am using in the above definition. I am saying that an interrupt is an *instance* of control flow interruption. Its cause is a not yet described event. We will be careful to distinguish the interrupt instance from the event which is its cause.

Pseudo-Interrupts

Many mechanisms are implemented which are said to interrupt a program but which don't quite fit the definition above. The most obvious example is the **supervisor call**. This is a machine instruction which transfers control in a protected manner to a subroutine or process which has privileged access to memory and other resources. The implementation of supervisor call instructions often uses methods similar to those used to handle interrupts. For instance, the vector table used for true interrupts might also be used for supervisor calls. Since a machine instruction must be coded in the program to effect control transfer, the "event" of executing the instruction cannot be said to cause an interrupt, because the control transfer is explicitly specified by the program and is not external to the program.

A slightly less obvious pseudo-interrupt is the processing of an *exception* (sometimes termed a *trap*). An **exception** is *a change in instruction sequence which occurs as a result of instruction execution, and which causes transfer of control to an exception handler not explicitly identified by the instruction.* The usual reason for an exception to be raised is that the instruction has determined that expected results cannot be delivered. A simple example is arithmetic overflow. Though control transfer to an exception handler is not explicitly specified by the instruction, the event causing the transfer comes from within the program, so that the transfer again cannot be considered an interrupt..

We will come back to supervisor calls and exceptions later. We are interested for now in interrupts, which are caused by events occurring external to the CPU hardware itself. In order to further our understanding of interrupts, we will examine how interrupts are usually handled in typical implementations.

How are interrupts typically handled by software?

Common terminology indicates that an interrupt handler is the software which receives control as a result of an interrupt, and that is how we will use the term here. Within a given computer architecture and operating system combination, design of an interrupt handler usually follows rules quite different from the design of any other software for that architecture and operating system. It is not clear whether an interrupt handler is a process, a subroutine, or something else. At least some part of an interrupt handler will have to be written in its target machine's assembly language, implying that high-level languages are not yet able to express the semantics of interrupt handling in a machine-independent way. A handler is severely limited by most operating systems as to what system services it may employ, or how it may communicate with processes. Portability of an interrupt handler is usually out of the question, even if most of it is written in a high-level language, partly because interrupt mechanisms vary so much from one processor architecture to the next. (I think that these are clues that the semantics of interrupts are not well understood.)

The processors in the Intel 80x86 family implement an interrupt as a subroutine call taken after completion of an instruction execution when an interrupting event has occurred. The subroutine call is asynchronous and invisible to whatever process might be running in the CPU at the time. The subroutine called (the "interrupt handler") must eventually return control to the point of its call, with the state of the interrupted process unchanged.

Implementation of interrupt handling as an invisible subroutine call seems inconsistent with the normal use of a subroutine which is expected to be quite visible to its caller. It must also be considered that the subroutine is not necessarily called on behalf of the interrupted process; therefore the subroutine may properly be a part of some other process or of no process at all.

Refining Interrupts

We have captured the essence of the term *interrupt* as it is normally used, but we have not yet been able to classify interrupt handlers as anything we are otherwise familiar with (such as subroutines or processes). Nor do we have a way of judging between the various interrupt mechanisms implemented in hardware to determine which ones do a better job than the others at "interrupting". We must step back and reexamine the purpose behind interrupt mechanisms.

Why do we want interrupts?

The primary motivation for providing an interrupt mechanism is to enhance multiprocessing within a single processor. Without an interrupt mechanism, a process waiting for an external event to occur would have to consume CPU cycles in a loop testing for the event—no other useful processing could be done—or would have to "poll" for the event periodically, requiring cooperation with other processes, and requiring CPU time for the polling. Interrupt mechanisms always include hardware to monitor for the occurrence of events concurrently with normal CPU instruction execution, so that the CPU can proceed with other processes not awaiting events. The mechanism is the means whereby the CPU is made aware that the awaited event has occurred, so waiting processes can be reactivated.

Where does an interrupting event come from?

We have identified above the common idea that an event external to the processor causes the interrupt. Most computers have hardware which channels signals from various sources into a single *interrupt request* line, and which keeps track of which signals are requesting an interrupt, and which ones have successfully caused an interrupt of the processor. An interrupt is therefore ultimately caused by one of some number of signals. Generalizing from this mechanism, we may say that a **signal** is *a single bit, the setting of which is an* **event** *and can cause an interrupt.* The unique property of a signal is that the event of its setting may be monitored by hardware concurrently with instruction execution. Processes can be blocked from consuming CPU cycles while awaiting any number of signal settings, thereby allowing non-blocked processes to run.

Redefining Interrupts

Let us revisit our earlier definition of *interrupt*, which accurately reflects the interrupt mechanism as it is usually implemented, but ignores the purpose of an interrupt.

Restricted Interrupt Handler

Traditionally, we program interrupt handlers so that, once the interrupt occurs, processing proceeds outside the normal process structure of the computer system in question. However, suppose we arbitrarily limit all interrupt handlers in a system to performing the same processing: readying processes awaiting the event which caused the interrupt. If no process is awaiting an event, the event is ignored. If a process which is made ready by an event has a higher priority than that of the process which was interrupted, then this process becomes the new running process. This approach is used in the Thoth operating system[6].

If the interrupt handling just described were the only interrupt processing implemented or possible in a computer system, then our definition of interrupt could be strengthened so that an **interrupt** would be *a change in CPU control flow in response to an external event, in order to ready processes awaiting the setting of the corresponding signal.* Normal processing would continue either with the interrupted process or a newly ready process, depending on the scheduling algorithm and process attributes such as priority.

The processing done by the universal interrupt handler would itself be uninterruptible by necessity, in order to finish processing an interrupt before commencing processing on the next interrupt to arrive. Processes scheduled to run on the processor would not run during this time, so that intermediate states of the uninterruptible software would be invisible to those processes. Now, any software whose execution is initiated by hardware and which runs without being able to be interrupted or preempted can be viewed as an extension of CPU hardware or firmware. From the point of view of all of the processes on the processor, there is no software which handles interrupts. This restricted definition of interrupt eliminates altogether the existence of programmed interrupt handlers. Instead, we have processes handling events. Our need for efficiently supporting hardware interrupts is satisfied without adding programmed control flow outside normal process control flow. Semantics need not be added to those normally used for expressing the action of concurrent processes.

Event Handling Within a Process

If an event is simply the setting of a bit which can be detected by hardware, and which can result in the readying of a process, then the effect of events on control flow within a process is entirely under the control of the process. Control would never be asynchronously changed *within* a process as the result of an event, although an interrupted process *might* be preempted because the setting of a signal made ready a process of higher priority. Processes could be programmed to behave deterministically with respect to the external events in which they were interested. That is to say, presented with the same sequence of external events, a process would react the same way each time it were executed.

A New Interrupt Mechanism

I propose the following semantics.

Let there be a subroutine called **wait_on_signal**, with two parameters: the signal_id and an alternative return point. A process calls **wait_on_signal** to test whether a signal bit is set. If it is, control returns to the alternative return point,with the caller's context unchanged. (By context I mean primarily the address of the stack frame.) If it is not set, control returns to the normal subroutine return point, but the calling process has been linked to the signal. A process may issue any number of **wait_on_signal** calls, provided that they are all issued from the same context. When a **wait_on_signal** call is issued, all signals for which **wait_on_signal** calls have been issued are examined, as well as the signal identified in the current call. If none of the signals are set at that time, control will return to the **wait_on_signal** subroutine's normal return point. However, control may be returned to any previous alternative return point for which a signal has been set.

Let there be a subroutine called **wait**, with no parameters. This subroutine makes the calling process not ready, taking the process off the ready queue and allowing another process to become the running process.

When a processor is interrupted as a result of an event, the universal interrupt handler identifies the signal which has just been set. It then readies all processes which are at that time waiting for that signal to be set. In other words, each process which has previously issued a **wait_on_signal** call for the signal just set, and which is currently not ready, will be made ready. Processes which have previously issued **wait_on_signal** calls for the signal just set, but which are already in a ready state, will be unaffected by the event. Processes which have not issued a **wait_on_signal** call for the signal just set will also be unaffected by the event. (Note that a signal is different from a binary semaphore because all processes waiting for it to be set will be readied when it is set, rather than one process each time it is set.)

At the completion of interrupt processing, processing would resume with the highest-priority ready process. This could be the process which was interrupted by the event, or it could be one of the processes readied by the interrupt handler.

Usage

A process which has to await the first of several possible asynchronous events would have a routine within which all of the `wait_on_signal` calls for those events would be coded. A simple `wait` call would be coded at the end of this list of `wait_on_signal` calls. As soon as any awaited signal were set, process execution would resume at the alternative return point associated with that signal.

In this way control flow is determined within a process by the process itself, and synchronously with the process's calls to `wait` and `wait_on_signal`. An interrupt never changes control flow within a process to a point not specified by that process.

There is no limitation on processing which can be done either between `wait_on_signal` calls or after an alternative return point has received control. Note that this is much more flexible than the `select` statement of Ada®[7]. All processing within the process occurs at the process's normal priority level, even if the process resumed execution as a result of a high-priority hardware event (the traditional hardware interrupt). A signal which becomes set cannot affect control flow within a process while that process 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. There is the added efficiency, important in some applications, that although the process must sequentially develop its list of awaited signals, control can be dispatched as a result of the setting of any of the signals awaited so far. A program can be coded to issue `wait_on_signal` calls first for the signals of greatest urgency, and can take whatever processing time is necessary to add to its list of awaited signals.

The above primitives allow a process to be programmed to reach a certain point after one of several arbitrary sequences of signals has been the first to occur. This will be shown in greater detail below.

Software Interrupts

Up to this point we have restricted our discussion to signals which are set by processes or hardware external to the CPU running the processes which are interested in these signals. If a signal is implemented as a bit in memory, then a subroutine called `set_signal` may be defined which sets a given signal bit, then readies all processes waiting for the signal to be set—a **software interrupt**. `set_signal` does not alter the instruction sequence of the process which calls it; however, it may cause preemption of the calling process by some process readied by it which has a higher priority than the calling process. If the calling process is preempted, its execution will be resumed when it is again the highest-priority ready process, and execution will be resumed at the return point of the `set_signal` call. The calling process will not be able to determine directly whether it was preempted during the `set_signal` call. Thus, a software interrupt, like a hardware interrupt, is invisible to the interrupted process.

Here is one of the great values of the proposed mechanism: a process may be programmed to await both software and hardware signals, without any special programming for the hardware signals. For instance, a machine control program could await a signal from a limit switch, the expiration of a (hardware or software) timer, the arrival of a message from another process, and the arrival of characters from a keyboard, without making any distinction as to whether the sources are hardware or software.

More subroutines

Most programs interested in events are interested in them each time they occur, not just the first time they occur. The awaiting is usually done inside a loop. When an event occurs, the part of the program is activated which deals with that event, and then control returns to the loop where that event and any other events are awaited again. Therefore, for the sake of efficiency, these semantics provide that each signal awaited remains awaited until the waiting is explicitly cancelled by an `ignore_signal` call identifying the signal. Even those signals which caused a control flow change within an awaiting process would remain awaited after the control flow change.

One more subroutine is needed: `reset_signal`. This does nothing more than reset the given signal bit. A process would never be preempted as a result of a `reset_signal` call.

Pseudo-Interrupts Revisited

We now have a definition of interrupt, and a description of hardware and software interrupts. We may now re-examine the so-called interrupts caused by supervisor calls and exceptions.

Supervisor Calls

The purpose of a supervisor call is to invoke some system service on behalf of the invoking process, and to do so while protecting system resources from access or modification by the invoking process. These goals are not directly related to multiprocessing goals, and therefore a supervisor call need not be implemented through an interrupt mechanism.

If a system service is to be provided by a process separate from the invoking user process, then it may make sense for a user process to set a signal on which a system process is waiting. Upon receipt of the signal, the system process would preempt the user process (presuming the system process's priority were higher than that of the user process) and begin servicing the user process's request for service. Protection mechanisms could be the same as those used between any processes in the system.

If a system service is to be provided by a system subroutine which is to run as part of the invoking user process, then a special instruction could be used which calls a subroutine through a restricted-access jump table. The instruction would provide the table index; the table address and its contents would be provided by the hardware and/or operating system. A subroutine called through this mechanism would run in the "privileged" mode of the system: it would have complete access to the invoking user's address space as well as system address space. Just before returning to the user a system subroutine would return to the "unprivileged" mode of operation.

Exceptions

The purpose of an exception is to alter control flow because an unusual ("exceptional") condition has arisen which is not provided for in the instruction sequence being executed. Like a supervisor call, this is not necessarily related to multiprocessing goals. Some high-level programming languages (such as Ada and C++ 3.0) provide ways of specifying actions to be taken when exceptions occur. When such actions are specified, it would seem logical that they be executed as part of the process in which the exceptions occurred. There are cases where this would not be possible. An example is a stack overflow exception, where it would be impossible for the process to proceed. In such a case, it would be better to suspend execution of the process in which the exception occurred, and set a signal on which a system process would be waiting. A system process would handle the exceptional condition.

Practical Implementation

It is perfectly feasible to program software interrupts of the type described above at the application level within most modern operating systems. In fact, in UNIX® operating systems even the `signal(2)` mechanism could be incorporated as a kind of source of hardware interrupts. Furthermore, within a given operating system individual interrupt handlers could be programmed as part of device drivers which communicate with application programs through software interrupts.

A universal interrupt handler of the type described above could be programmed to handle all hardware interrupts for most of the processor architectures in use today. The exact processing to be performed by such a handler would depend heavily on the process scheduling and priority scheme used by the operating system. We will examine how such an interrupt handler would have to be implemented within a preemptive priority process scheduling system in order to preserve responsiveness to real-time events. We will start by examining the functioning of the handler to implement the semantics already described. We will then restrict the semantics of hardware interrupts in order to arrive at an efficient implementation.

Preemptive Priority Scheme

For the purposes of our discussion, we will make the following assumptions about processes and scheduling. At any moment, each process in the system is in one of several states with respect to scheduling; the two in which we are particularly interested are **`not_ready`** and **`ready`**. Each process also has an attribute called

priority. All processes which are in a ready state are linked together in priority order on the **ready queue**, with the highest-priority process at the head. The scheduler guarantees that the highest priority ready process is also the **running** process, that is, the process which is currently using the CPU for execution. The running process remains the running process until either it becomes not_ready (possibly as a result of executing a **wait** call), or it is **preempted** by a process which enters the ready state and which has a higher priority than the running process. If a process becomes not_ready, the highest priority ready process, which is at the head of the ready queue, becomes the new running process. If a process is preempted, it is placed at the head of the ready queue as the next available process for the CPU to run.

Unrestricted Hardware Interrupt Handling

If an interrupt request were accepted by a processor, normal execution of the running process would be suspended, and control flow would pass to the universal interrupt handler. The handler would identify the signal which was just set, and identify all of the not_ready processes waiting for the setting of the signal. It would make each of these processes ready, and place each one on the ready queue in priority order. It would then determine whether a newly-readied process should preempt the interrupted process. If not, the processor would simply resume execution of the interrupted process. If a newly-readied process were to become the running process, the processor would save the context of the interrupted process in that process's control structure, and restore the context of the newly-readied process.

Without further refinements, this scheme would be inefficient and could render a computer system useless for handling real-time events. In real-time systems, the speed with which a process can gain control of the CPU after an event has occurred is critical. The responsiveness of a system also depends on the amount of time spent scheduling processes rather than executing them. Placing processes on a ready queue could involve insertion into an ordered linked list, which can be costly if the list is long. This scheduling time itself would be variable, since it would depend on the length of the ready queue, and that would be unpredictable at the time of the interrupt. Finally, all of this processing would take place with the CPU in a state where no other interrupts could occur, adding to the delay in processing signals from events.

Limiting the number of processes waiting for a signal

The signal mechanism has been described as generally as possible, with no limitations on the number of processes which may set a signal bit or which may wait for a bit to be set. If we limit the number of processes which may wait on an external hardware signal to one, then zero or one processes will be readied when an interrupt request is accepted. If a process were readied, it might preempt the `running` process or it might have to be put on the `ready queue`. Each of these two cases must be considered.

In the case of preemption a known amount of processing will be needed to replace the `running` process on the `ready queue`. This is because it is known that the `running` process is the highest priority `ready` process except for the newly `ready` process, and therefore may simply be linked to the head of the `ready queue`. A fixed amount of processing will also be needed to make the newly `ready` process the `running` process. This newly `ready` process will not have to be unlinked from some other point in the `ready queue`, since it was known not to be `ready` before the interrupt. Therefore, an interrupt resulting in preemption can be handled in a fixed amount of processing time.

If a process is made `ready` by the interrupt handler but does not preempt the `running` process, it will have to be linked into the `ready queue` in `priority` sequence. The time necessary to do this is a function of the newly `ready` process's new position in the `ready queue`, and is therefore variable. If we can eliminate this variability, then we can depend on the fact that each interrupt requires a fixed amount of processing, whether or not it results in preemption. We will deal with this problem below.

Interrupt Priority

Software interrupts derive their `priority` from the priorities of the processes setting and waiting on signals. If we examine software interrupts, we see that the **urgency** of an event is related to the `priority` of the process setting the signal, but the speed of **response** to the event is related to the `priority` of the process waiting for the signal to be set, relative to the `priority` of the `running` process. A signal will be set when the process setting the signal is the highest priority `ready` process in the system, but a process awaiting that signal will be able to respond to it only when it is the highest priority `ready` process. Thus, a process which sets a signal which is awaited by a process of equal or lower `priority` will not be preempted, while a process which sets a signal which is awaited by a process of higher `priority` will be preempted.

Many hardware interrupt mechanisms can associate a `priority` with each signal which can cause an interrupt, to enhance responsiveness to events of varying urgency. The CPU can have a `priority` attribute which must be exceeded by the `priority` of a signal in order for the setting of that signal to cause an interrupt. If this `priority` attribute is equal to that of the `running` process, then the CPU will accept an interrupt request only if the event's `priority` is greater than that of the `running` process. The `running` process, however, will not be preempted unless the `priority` of the process made `ready` by the interrupt is greater than that of the `running` process.

Let us require that a process awaiting a particular event must run with `priority` equal to or greater than that of the event. Since the CPU will only accept an interrupt request if the event's `priority` is greater than that of the `running` process, this requirement guarantees that acceptance of an interrupt request by a CPU will result in the readying of a process with `priority` greater than that of the `running` process, and will therefore result in preemption of the `running` process.

Thus we have the following rules for a hardware interrupt scheme using this mechanism. Zero or one processes may wait at any one time for a signal associated with an external hardware event. An interrupt request is accepted by a 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. The `priority` of a process waiting for an external event must be greater than or equal to the (hardware) `priority` of that event, thus guaranteeing preemption of the `running` process upon acceptance of the interrupt request.

With these restrictions on hardware interrupts, the processing done by a CPU upon acceptance of an interrupt request will always be no more than to save the context of the `running` process, link the `running` process to the head of the `ready queue`, and restore the context of the newly `ready` process. There would therefore always be a fixed amount of processing necessary to reschedule processes when a hardware interrupt request was accepted. Since the universal interrupt handler would run uninterruptibly, this scheme fixes and minimizes the amount of time the CPU is unavailable for processing other interrupt requests. This makes it practical not only to implement such a universal interrupt handler as part of the kernel of a real-time operating system, but also to consider implementing this kind of interrupt handling in the CPU hardware or firmware itself.

These restrictions do not violate the semantics of the signal as described above, and need not be applied to software interrupts. In fact, if it is necessary for more than one process to await a single hardware signal, one process can be designed to await the signal, then set a software signal on which multiple processes are waiting.

More Usage

It may be desirable to design a process which would wait for a number of events of varying degrees of urgency. Such a process could set its `priority` equal to the highest `priority` of the events for which it was waiting, before it began a series of **`wait_on_signal`** calls. After receiving control at some point because a particular signal had been set, the process would set its `priority` equal to that of the corresponding event, then respond to the event. This would guarantee that receipt by the CPU of an interrupt on behalf of this process would result in the preemption of the `running` process, although the new `running` process may itself be preempted later if it lowers its own `priority`.

Other Considerations

It is beyond the scope of this paper, but still relevant to this topic, to consider other issues relating to the awaiting of events and the scheduling of processes. It would be ideal if processes could await any combination of things equally and without restriction: not only hardware interrupts and software interrupts, but also the arrival of messages, the availability of semaphores, the termination of processes, etc. The signal mechanism I have described could be used as the underlying mechanism for notification of all of these events, or a more general **`wait_on_event`** routine could be designed. The important goal is that software can be written at the application level to deal with asynchronous events in a system in an unrestricted manner.

Additional refinements to the scheduling mechanism can be made to further reduce the overhead in readying and scheduling processes.

Processes which have been made ready but which do not have sufficient priority to preempt the running process can be placed on a pending queue. Scheduling these processes can be deferred until the running process waits or lowers its priority. This reduces the likelihood that time will be taken at a high priority to schedule processes of lower priority. I am indeed designing such a scheduling system at this time to run within the MS-DOS operating system.

Benefits

The interrupt mechanism proposed above provides the following benefits:

1. There is no longer a special class of software called an "interrupt handler". Processes responding to events external to the processor can use system services in exactly the same way as any other processes.

2. All processes in a system can use the same semantics for sending and receiving signals amongst themselves and externally to the processor.

3. Processes responding to events external to the processor can be scheduled in exactly the same way as other processes. These processes can dynamically change their priorities, and can be pre-empted by higher priority processes.

4. A process can simultaneously wait for any number of internal or external events with the same mechanism and without distinguishing between internal and external events.

5. An external interrupt request is never accepted until it is known that the interrupt will result in preemption of the running process. This improves the efficiency of handling interrupts.

6. Control flow within a single process is completely under the control of that process's code. No asynchronous control transfers occur as the result of asynchronous events.

7. During development of processes designed to communicate with hardware not yet available, a debugging process can provide the signals expected from hardware with no changes in the process under development.

8. If a universal interrupt handler implementing these semantics is written for several processor architectures, software handling external events can be moved portably from architecture to architecture with no changes necessary due to different interrupt mechanisms.

References

[1]International Business Machines Corp. IBM System/370 Principles of Operation. Poughkeepsie, NY: IBM Corp., 1973, p.68.

[2]National Semiconductor Corporation. *Introduction to the NS16000 Architecture*. Santa Clara, California: National Semiconductor Corporation, 1983, p. 27.

[3]Motorola, Inc. *MC68020 Technical Summary.* Phoenix, Arizona: Motorola, Inc., 1986, p. 9.

[4]Intel Corporation. *80386 Programmer's Reference Manual.* Santa Clara, California: Intel Corporation, 1987, p. 9-1.

[5]SPARC International, Inc. *The SPARC Architecture Manual, Version 8.* Englewood Cliffs, NJ: Prentice Hall, 1992, p. 14.

[6]Cheriton, David R. *The Thoth System: Multi-Process Structuring and Portability.* New York: North-Holland, The Computer Science Library, Operating and Programming Systems Series, #8, 1982, pp. 36-37.

[7]Department of Defense. *Military Standard: Ada Programming Language*. Washington, D.C.: Superintendent of Documents, U.S. Government, ANSI/MIL-STD-1815A-1983, pp. 9-12—9-16.