Setting Interrupt Priorities in Software via Interrupt Queueing

Geoff Collyer Bell Laboratories

ABSTRACT: When hardware interrupt priorities don't match the needs of software, operating system designers often just suffer in silence. We describe an alternative here: simulating the hardware priority interrupt queueing mechanism in software, but assigning the (software) interrupt priorities as we wish. This was done on an AMD 29200 microcontroller [Advanced Micro Devices 1994] which has effectively only two interrupt levels: the clock and everything else.

1. Introduction

An *interrupt* is an asynchronous exception or processor trap, typically generated by a peripheral device to notify the CPU of the completion of some I/O operation. This exception typically produces a trap to supervisor (or kernel) mode at an address reserved for interrupts, and leaves some indication of what caused the interrupt in the processor state. In UNIX^{TM}-like systems, the *first-level interrupt handler* (FLIH) is responsible for saving any machine state that may be altered and not saved by the C language call sequence, determining the interrupt's cause, calling the appropriate C language [Kernighan & Ritchie 1978] *interrupt service routine* or *interrupt handler*, restoring the saved machine state, dismissing the hardware interrupt, and resuming the interrupted program, so as to make the interrupt *transparent*. An interrupt service routine is a subroutine executed in response to a specific interrupt. It typically notes the completion of the most recent I/O operation for the device that generated the interrupt and may initiate the next operation, if one is pending, or it may signal non-interrupt code that the interrupt has occurred. The time taken to begin executing the interrupt service routine is called *interrupt response time*.

Many computers have an interrupt priority scheme, in which high-priority devices can interrupt the service routines of lower-priority devices. Priority is typically determined by hardware, based on some combination of device type and physical location on the bus (e.g. the DEC PDP-11 family). Some machines (e.g., later PDP-11s) [Digital 1976] even have some form of *programmable interrupt request* (PIRQ): a means for the operating system to simulate a hardware interrupt at a given priority.

A fairly typical arrangement of priorities is this: the clock has top priority and most everything else shares the next priority level down [Ritchie 1979]. For general use, this is usually fine, but when there is a device requiring very fast interrupt response (or more than one such device!), one would really like to have such devices get higher-than-normal priority. Devices containing substantial data buffering generally do not require fast interrupt response (though it remains desirable), but for devices with little or no buffering, the usual consequence of late interrupt response is lost input data or some physical manifestation such as a ruined CD ROM or flickering video display.

We have implemented software queueing of interrupts within AT&T's HomeCenter™, a television set-top box that does not require vast bandwidth to the home. HomeCenter uses an inexpensive AMD 29200 RISC processor with a single-process operating system borrowing from Plan 9™ [Pike et al. 1990] and UNIX [Ritchie & Thompson 1978]. Though it runs at 16MHz, our AMD 29200 achieves only about 5 MIPS, largely because it lacks caches and hence is forever accessing memory. Its register windows can also hurt interrupt performance if we need to spill the registers in a service routine. The HomeCenter video interface needs to be refreshed at 60Hz with *extremely* fast response to its interrupt (well under a millisecond). We chose to give up a little performance in exchange for much faster interrupt response for a few devices and our choice of interrupt priorities.

Register windows are a scheme for using lots of general-purpose registers that originated at UC Berkeley [Patterson & Ditzel 1980]. The idea is to treat a large set of general registers as a cache of the stack top. When the stack shrinks sufficiently to require addressing outside the general registers, the general registers are *filled* by copying from the top of the *register stack*. When the stack grows sufficiently to require more general registers, the general registers are *spilled* by copying them to the top of the register stack.

Software queueing of interrupts seems to be alluded to in the literature, particularly the older literature [Gear 1974], but detailed expositions are lacking. (In particular, Mach 3.0 queues at most one interrupt per priority level [they call these queued interrupts *interrupt continuations*] and the objective is greater overall efficiency rather than faster interrupt response [Stodolsky et al. 1993]. We would be unable to provide fast enough interrupt response using the Mach scheme, since it masks interrupts once an interrupt is received during a critical section.) RISC machines and low-cost microcontrollers may revive this venerable technique.

2. *Overview*

The idea is to emulate the usual hardware priority interrupt scheme in software. Traditionally, processor status words contain a *processor level* which is the highest interrupt priority for which interrupts are currently being deferred. In HomeCenter, except for very brief intervals of software-interrupt queue maintenance, the system runs with all hardware interrupts enabled at all times.

As hardware interrupts are received, they are queued by software priority and arrival order. They are dequeued and run (by setting the software's virtual processor level or 'VPL' to that in the queue entry and invoking the interrupt service routine in the queue entry) by the `spln` routines [Ritchie 1979] as the VPL drops. Traditionally these routines are used to protect critical sections of code that manipulate data structures also manipulated by interrupt service routines.The `spln` routines now set the VPL to *n*, rather than actually changing the processor level, to disable only interrupts at or below priority *n*. Dequeueing is a bit tricky. In particular, clock interrupt routines may drop VPL when doing lengthy computations (e.g., arbitrary time-out callbacks) and may expect to be interrupted by later clock interrupts.

3. Caveat

This may all sound straightforward or obvious, but getting the details right is important and may involve intimate knowledge of the hardware. Some years ago, an acquaintance with an Interdata 8/32 [Interdata 1976] running Interdata's "Edition Seven" UNIX system asked around to see if anyone could get the unbuffered Interdata UARTs (serial ports) to accept input any faster than 1200 bps (for *uucp*). The author implemented a first cut at the scheme described here, which did permit faster UART input but also ruined the Interdata file systems (which had all been backed up just before the test). Probably lack of familiarity with the Interdata hardware had caused some subtle point to be overlooked. (The hardware 'helped' by providing an interrupt queue.) In any case, this is the only kernel work by the author (which covers the years from 1978 to the present) that has ever caused file system damage. Caveat implementor.

4. Implementation

Most devices have a status register containing a *ready bit* that indicates that the device is ready to accept a new command. The transition from not-ready to ready can often be made to generate at least one interrupt. We assume that all interrupts are edge-sensitive: exactly one interrupt is generated by the transition of a device's ready bit from not-ready to ready. In some cases we condition the hardware to make it so. Level-sensitive interrupts, wherein interrupts are generated continuously (processor level permitting) while the device is ready, have not been dealt with. (They can be handled by having a small device-specific routine for each

```
#define CONTROLLER ((struct diskregs *)0176500)
...
/* NB: disk interrupts are inhibited while this runs */
diskintr()
{
    struct diskregs *rp = CONTROLLER;
    struct diskstate *dp = &disks[rp->unit];

    mark previous transfer (dp->queue->head) done;
    if (dp->queue not empty)
          initiate next request on controller rp;
}
```

Figure 1. Disk interrupt service routine pseudocode.

level-sensitive interrupt, that turns off that hardware interrupt, thereby emulating an edge-sensitive interrupt. Something similar was done in Modula [Wirth 1977], for slightly different reasons.)

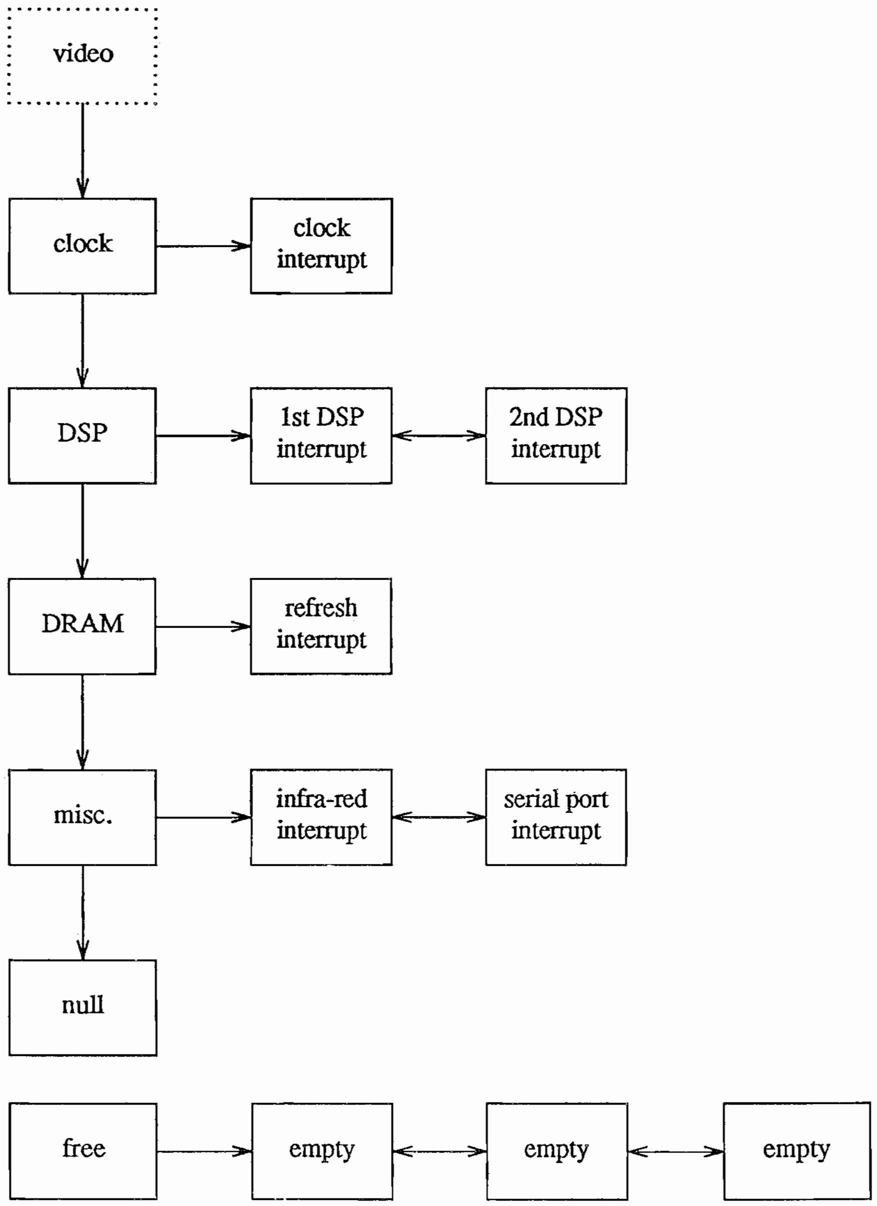
A traditional interrupt service routine written in C might look something like the pseudo-code in Figure 1.

Traditionally, uniprocessor UNIX kernels use elevated processor level to defer or coalesce interrupts during critical sections and interrupt service routines. We have made the processor level seen by the kernel virtual, with some subtle ramifications described below.

The interrupt queueing scheme in use in HomeCenter has evolved somewhat. Initially a single singly-linked queue of interrupts was maintained, sorted by software priority (in decreasing order) and then by arrival order. This turned out to be inadequate to provide interruptible clock interrupts and was in general fairly costly and clumsy. The current scheme uses doubly-linked queues: a free queue and a queue for each software priority. The doubly-linked queues permit fast queue maintenance in constant time. Figure 2 shows an example snapshot of the interrupt queues.

The code executed in response to a hardware interrupt must call the common routine `interrupt`, with a software priority and interrupt handler, for each interrupt to be queued, before dismissing the hardware interrupt. `interrupt` queues a representation of the interrupt:

handler	the interrupt service routine's address,
swpri	the software priority (redundant since the queue determines priority),
flags	running or not running the service routine currently, and
serial	a unique number used in debugging.



The DSP is HomeCenter's digital signal processor, an AT&T 1603 or 1604.

Figure 2. Example interrupt queues.

```
idle()
{
    int s = spl0();

    put the processor into a wait or idle mode until the next interrupt;
    splx(s);
}
```

Figure 3. Usual idle pseudocode.

A queued-interrupt structure is removed from the free queue, filled in and appended to the queue for this software priority. After queueing one or more interrupts, `intrrunqs` should be called to service, in decreasing order by software interrupt priority, any interrupts not already being serviced and queued at software interrupt priority higher than the VPL. `intrrunqs` is also called by the `spln` routines.

`intrrunqs` scans the interrupt queues repeatedly, each time from top priority down to one above the current VPL, calling `run1intr` on the non-empty queue of highest priority until `intrrunqs` gets to the current VPL or `run1intr` finds a queue with an interrupt currently being serviced.

`run1intr` skips any initial queued interrupts with the RUNNING flag set and returns with a distinctive return code if there are no other interrupts in this queue (i.e. at this software interrupt priority). This permits interrupt service routines that drop VPL to be interrupted. If there is a non-RUNNING interrupt in the queue, the first such is chosen, its RUNNING bit is set, VPL is set to the interrupt's software priority, hardware interrupts are *enabled*, and the interrupt's handler is called. Upon return from the handler, hardware interrupts are disabled, VPL is restored, the interrupt is removed from this queue (and the structure zeroed to avoid accidents) and the structure is appended to the free queue.

If there is a routine called when there is no work to do and to wait for the next interrupt, it should be modified to suit this scheme. Our routine (and UNIX's) is called *idle*. The usual pseudo-code is in Figure 3. This code will idle the processor even if there is a queued interrupt, so we alter this pseudo-code to that in Figure 4.

The whole scheme is a little over-engineered, but caution was considered wise; see *Caveat* above.

Running with hardware interrupts enabled exposes code that expected to compress multiple interrupts into a single one by running at elevated processor level, so code may have to be modified to be robust against multiple interrupts where

```
idle()
{
   int doidle = 0, s;

   disable interrupts; /* get a consistent view */
   if (free-interrupt queue full)
      doidle = 1; /* nothing queued: wait for interrupt */
   s = spl0(); /* run sw interrupts, if any queued */
   if (doidle)
      put the processor into a wait or idle mode until the next interrupt;
   splx(s);
   restore interrupts;
}
```

Figure 4. Revised idle pseudocode.

previously only one was possible.

One also needs to beware of ill-behaved devices (e.g., attached processors) whose interrupt service routines drop software 'processor level': a flurry of interrupts can grow the (kernel) stack enormously. We have found it necessary to quickly dismiss all but the first of a series of interrupts indicating that some status bit is (spuriously) flipping on and off rapidly.

Thus, in HomeCenter, pseudo-code for the above-cited disk interrupt would be more like Figure 5 (including pseudo-code for the interrupt queueing).

```
run1intr(q)
struct intrq *q;
{
    struct qdintr *qi = q->head;

    if (qi == 0)
         return 1;
    remove qi from q;
    set VPL to qi->swpri;
    enable interrupts;
    (*qi->handler)();
    disable interrupts;
    restore VPL;
    put qi on the free queue;
    return 0;
}
```

Figure 5. Revised disk interrupt pseudocode. (*continued on next page*)

```
intrrunqs()
{
    struct intrq *q;

    do {
        q = 0;
        scan interrupt queues from highest to lowest priority
           (VPL) and select a non-empty queue, q, if any;
    } while (q != 0 && run1intr(q) == 0);
}

interrupt(pri, handler)
int pri;
void (*handler)();
{
    struct intrq *q = intrqs[pri];
    struct qdintr *qi;

    remove qi from free queue;
    if (free queue exhausted)
        panic();
    qi->swpri = pri;
    qi->handler = handler;
    add qi to q;
}
...
#define CONTROLLER ((struct diskregs *)0176500)
...
diskintrwork()
{
    struct diskregs *rp = CONTROLLER;
    struct diskstate *dp = &disks[rp->unit];

    mark previous transfer (dp->queue->head) done;
    if (dp->queue not empty)
        initiate next request on controller rp;
}
diskintr()
{
    interrupt(Diskpri, diskintrwork);
    intrrunqs();
}
```

Figure 5. (*continued*) Revised disk interrupt pseudocode.

```
run1intr(q)
struct intrq *q;
{
    ...
    (*qi->handler)(qi->ctrlr);
    ...
}
...
interrupt(pri, handler, ctrlr)
int pri, ctrlr;
void (*handler)();
{
    ...
    qi->swpri = pri;
    qi->handler = handler;
    qi->ctrlr = ctrlr;
    add qi to q;
}
...
diskintrwork(ctrlr)
int ctrlr;
{
    struct diskregs *rp = CONTROLLER[ctrlr];
...
}

diskintr(ctrlr)
int ctrlr; /* from FLIH */
{
    interrupt(Diskpri, diskintrwork, ctrlr);
    intrrunqs();
}
```

Figure 6. Projected disk interrupt pseudocode.

HomeCenter is unusual in having only one of each peripheral; general-purpose computers are likely to, at least potentially, have multiple device controllers of the same type (e.g., serial ports). In such cases, the FLIH generally provides an argument to the interrupt handlers containing the number of the controller interrupting. To accommodate multiple controllers, we would need to add a 'controller' argument to `interrupt`, the queued-interrupt structure (`qdintr`) and the interrupt handlers as in Figure 6.

5. Bonuses

With this scheme in place, several features fell out fairly naturally. A video interface requiring very fast response, but very simple processing, was dealt with by avoiding the interrupt queue altogether, and just calling its interrupt service routine directly from `interrupt`.

A programmable interrupt request (PIRQ) facility also fell out. This is used to perform *software refresh* of DRAM at middling interrupt priority. Normally hardware arranges to refresh DRAM (which will decay if not read roughly every 8 ms.) transparently, but in our case the hardware refresh of the 29200 produces occasional video glitches (part of a scan line turns white), so we turn off the hardware refresh and simulate it, at some cost (5% of the CPU per megabyte of DRAM refreshed), in software.

6. Costs

There is obviously additional CPU overhead to queueing interrupts and there is memory associated with the queue, which must be large enough to accommodate infrequent surges. The `spln` routines must scan the interrupt queues when 'processor level' drops, but this is quick. One would not use this technique were the hardware interrupt priorities appropriate to the task at hand, but in the absence of such hardware, it has its uses.

7. Conclusion

Software interrupt queueing costs a little overhead, but frees system designers from bad choices of hardware interrupt priorities and provides other benefits, such as a programmable interrupt request facility. Careful implementation yields a system which needs little modification to device drivers while meeting unusual needs easily.

Acknowledgments

Henry Spencer, Mark Moraes and three anonymous reviewers critiqued drafts of this paper, improving it thereby. Barry Shein helped me search the databases of USENIX papers. Dave Sherman allowed me to experiment with my first cut of this technique on his Interdata machine.

References

1. Advanced Micro Devices, *Am29200 and Am29205 RISC Microcontrollers User's Manual*, 1994.
2. Digital Equipment Corporation, *pdp11/70 processor handbook*, 1976.
3. C. William Gear, *Computer Organization and Programming, 2nd ed.*, McGraw-Hill (1974).
4. Interdata Inc, *Model 8/32 Processor User's Manual*, January 1976.
5. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1978).
6. A. M. Lister, *Fundamentals of Operating Systems, Second Edition*, Macmillan (1979).
7. D. Patterson and D. R. Ditzel, The Case for the Reduced Instruction Set Computer, *Computer Architecture News* **8**(7) (1980).
8. Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey, Plan 9 from Bell Labs, *Proc. of the Summer 1990 UKUUG Conf., London*, pages 1–9 (July, 1990).
9. D. M. Ritchie and K. Thompson, The UNIX Time-Sharing System, *Bell Sys. Tech. J.* **57**(6), pages 1905–1929 (1978).
10. Dennis M. Ritchie, The UNIX I/O System, *UNIX Programmer's Manual, Seventh Edition* (January, 1979).
11. Daniel Stodolsky, J. Brad Chen, and Brian Bershad, Fast Interrupt Priority Management in Operating System Kernels, pages 105–110, *Microkernels and Other Kernel Architectures Symposium II*, USENIX, San Diego, CA (Sept. 20–21, 1993).
12. N. Wirth, Design and Implementation of Modula, *Software—Practice & Experience* **7**, pages 67–84 (1977).