Interrupts

2.5.2-rmk5:

This is the first kernel that contains a major shake up of some of the major architecture-specific subsystems.

Firstly, it contains some pretty major changes to the way we handle the MMU TLB. Each MMU TLB variant is now handled completely separately - we have TLB v3, TLB v4 (without write buffer), TLB v4 (with write buffer), and finally TLB v4 (with write buffer, with I TLB invalidate entry). There is more assembly code inside each of these functions, mainly to allow more flexible TLB handling for the future.

Secondly, the IRQ subsystem.

The 2.5 kernels will be having major changes to the way IRQs are handled. Unfortunately, this means that machine types that touch the irq_desc[] array (basically all machine types) will break, and this means every machine type that we currently have.

Lets take an example. On the Assabet with Neponset, we have:

```
           GPIO25                IRR:2
SA1100 ------------> Neponset -----------> SA1111
                                 IIR:1
                               -----------> USAR
                                 IIR:0
                               -----------> SMC9196
```

The way stuff currently works, all SA1111 interrupts are mutually exclusive of each other - if you're processing one interrupt from the SA1111 and another comes in, you have to wait for that interrupt to finish processing before you can service the new interrupt. Eg, an IDE PIO-based interrupt on the SA1111 excludes all other SA1111 and SMC9196 interrupts until it has finished transferring its multi-sector data, which can be a long time. Note also that since we loop in the SA1111 IRQ handler, SA1111 IRQs can hold off SMC9196 IRQs indefinitely.

The new approach brings several new ideas...

We introduce the concept of a "parent" and a "child". For example, to the Neponset handler, the "parent" is GPIO25, and the "children"d are SA1111, SMC9196 and USAR.

We also bring the idea of an IRQ "chip" (mainly to reduce the size of the irqdesc array). This doesn't have to be a real "IC"; indeed the SA11x0 IRQs are handled by two separate "chip" structures, one for GPIO0-10, and another for all the rest. It is just a container for the various operations (maybe this'll change to a better name). This structure has the following operations:

```
struct irqchip {
        /*
         * Acknowledge the IRQ.
         * If this is a level-based IRQ, then it is expected to mask the IRQ
         * as well.
         */
        void (*ack)(unsigned int irq);
        /*
         * Mask the IRQ in hardware.
         */
        void (*mask)(unsigned int irq);
        /*
         * Unmask the IRQ in hardware.
         */
        void (*unmask)(unsigned int irq);
        /*
         * Re-run the IRQ
         */
        void (*rerun)(unsigned int irq);
        /*
         * Set the type of the IRQ.
         */
        int (*type)(unsigned int irq, unsigned int, type);
};
```

ack
- required. May be the same function as mask for IRQs handled by do_level_IRQ.

mask
- required.

unmask
- required.

rerun
- optional. Not required if you're using do_level_IRQ for all IRQs that use this 'irqchip'. Generally expected to re-trigger the hardware IRQ if possible. If not, may call the handler directly.

type
- optional. If you don't support changing the type of an IRQ, it should be null so people can detect if they are unable to set

the IRQ type.

For each IRQ, we keep the following information:

- "disable" depth (number of disable_irq()s without enable_irq()s)
- flags indicating what we can do with this IRQ (valid, probe, noautounmask) as before
- status of the IRQ (probing, enable, etc)
- chip
- per-IRQ handler
- irqaction structure list

The handler can be one of the 3 standard handlers - "level", "edge" and "simple", or your own specific handler if you need to do something special.

The "level" handler is what we currently have - its pretty simple. "edge" knows about the brokenness of such IRQ implementations - that you need to leave the hardware IRQ enabled while processing it, and queueing further IRQ events should the IRQ happen again while processing. The "simple" handler is very basic, and does not perform any hardware manipulation, nor state tracking. This is useful for things like the SMC9196 and USAR above.

So, what's changed?

1. Machine implementations must not write to the irqdesc array.
2. New functions to manipulate the irqdesc array. The first 4 are expected to be useful only to machine specific code. The last is recommended to only be used by machine specific code, but may be used in drivers if absolutely necessary.

   set_irq_chip(irq,chip)
   : Set the mask/unmask methods for handling this IRQ

   set_irq_handler(irq,handler)
   : Set the handler for this IRQ (level, edge, simple)

   set_irq_chained_handler(irq,handler)
   : Set a "chained" handler for this IRQ - automatically enables this IRQ (eg, Neponset and SA1111 handlers).

   set_irq_flags(irq,flags)
   : Set the valid/probe/noautoenable flags.

   set_irq_type(irq,type)
   : Set active the IRQ edge(s)/level. This replaces the SA1111 INTPOL manipulation, and the set_GPIO_IRQ_edge() function. Type should be one of IRQ_TYPE_xxx defined in <linux/irq.h>

3. set_GPIO_IRQ_edge() is obsolete, and should be replaced by set_irq_type.
4. Direct access to SA1111 INTPOL is deprecated. Use set_irq_type instead.
5. A handler is expected to perform any necessary acknowledgement of the parent IRQ via the correct chip specific function. For instance, if the SA1111 is directly connected to a SA1110 GPIO, then you should acknowledge the SA1110 IRQ each time you re-read the SA1111 IRQ status.
6. For any child which doesn't have its own IRQ enable/disable controls (eg, SMC9196), the handler must mask or acknowledge the parent IRQ while the child handler is called, and the child handler should be the "simple" handler (not "edge" nor "level"). After the handler completes, the parent IRQ should be unmasked, and the status of all children must be re-checked for pending events. (see the Neponset IRQ handler for details).
7. fixup_irq() is gone, as is *arch/arm/mach-*/include/mach/irq.h*

Please note that this will not solve all problems - some of them are hardware based. Mixing level-based and edge-based IRQs on the same parent signal (eg neponset) is one such area where a software based solution can't provide the full answer to low IRQ latency.