The padata parallel execution mechanism
Last updated for 2.6.34

Padata is a mechanism by which the kernel can farm work out to be done in parallel on multiple CPUs while retaining the ordering of tasks. It was developed for use with the IPsec code, which needs to be able to perform encryption and decryption on large numbers of packets without reordering those packets. The crypto developers made a point of writing padata in a sufficiently general fashion that it could be put to other uses as well.

The first step in using padata is to set up a padata_instance structure for overall control of how tasks are to be run:

```
#include <linux/padata.h>

struct padata_instance *padata_alloc(const struct cpumask *cpumask,
                                     struct workqueue_struct *wq);
```

The cpumask describes which processors will be used to execute work submitted to this instance. The workqueue wq is where the work will actually be done; it should be a multithreaded queue, naturally.

There are functions for enabling and disabling the instance:

```
void padata_start(struct padata_instance *pinst);
void padata_stop(struct padata_instance *pinst);
```

These functions literally do nothing beyond setting or clearing the "padata_start() was called" flag; if that flag is not set, other functions will refuse to work.

The list of CPUs to be used can be adjusted with these functions:

```
int padata_set_cpumask(struct padata_instance *pinst,
                       cpumask_var_t cpumask);
int padata_add_cpu(struct padata_instance *pinst, int cpu);
int padata_remove_cpu(struct padata_instance *pinst, int cpu);
```

Changing the CPU mask has the look of an expensive operation, though, so it probably should not be done with great frequency.

Actually submitting work to the padata instance requires the creation of a padata_priv structure:

```
struct padata_priv {
    /* Other stuff here... */
    void                    (*parallel)(struct padata_priv *padata);
    void                    (*serial)(struct padata_priv *padata);
};
```

This structure will almost certainly be embedded within some larger structure specific to the work to be done. Most its fields are private to padata, but the structure should be zeroed at initialization time, and the parallel() and serial() functions should be provided. Those functions will be called in the process of getting the work done as we will see momentarily.

The submission of work is done with:

```
    int padata_do_parallel(struct padata_instance *pinst,
                           struct padata_priv *padata, int cb_cpu);
```

The pinst and padata structures must be set up as described above; cb_cpu specifies which CPU will be used for the final callback when the work is done; it must be in the current instance's CPU mask. The return value from padata_do_parallel() is a little strange; zero is an error return indicating that the caller forgot the padata_start() formalities. -EBUSY means that somebody, somewhere else is messing with the instance's CPU mask, while -EINVAL is a complaint about cb_cpu not being in that CPU mask. If all goes well, this function will return -EINPROGRESS, indicating that the work is in progress.

Each task submitted to padata_do_parallel() will, in turn, be passed to exactly one call to the above-mentioned parallel() function, on one CPU, so true parallelism is achieved by submitting multiple tasks. Despite the fact that the workqueue is used to make these calls, parallel() is run with software interrupts disabled and thus cannot sleep. The parallel() function gets the padata_priv structure pointer as its lone parameter; information about the actual work to be done is probably obtained by using container_of() to find the enclosing structure.

Note that parallel() has no return value; the padata subsystem assumes that parallel() will take responsibility for the task from this point. The work need not be completed during this call, but, if parallel() leaves work outstanding, it should be prepared to be called again with a new job before the previous one completes. When a task does complete, parallel() (or whatever function actually finishes the job) should inform padata of the fact with a call to:

```
    void padata_do_serial(struct padata_priv *padata);
```

At some point in the future, padata_do_serial() will trigger a call to the serial() function in the padata_priv structure. That call will happen on the CPU requested in the initial call to padata_do_parallel(); it, too, is done through the workqueue, but with local software interrupts disabled. Note that this call may be deferred for a while since the padata code takes pains to ensure that tasks are completed in the order in which they were submitted.

The one remaining function in the padata API should be called to clean up when a padata instance is no longer needed:

```
    void padata_free(struct padata_instance *pinst);
```

This function will busy-wait while any remaining tasks are completed, so it might be best not to call it while there is work outstanding. Shutting down the workqueue, if necessary, should be done separately.