The padata parallel execution mechanism

Date: May 2020

Padata is a mechanism by which the kernel can farm jobs out to be done in parallel on multiple CPUs while optionally retaining their ordering.

It was originally developed for IPsec, which needs to perform encryption and decryption on large numbers of packets without reordering those packets. This is currently the sole consumer of padata's serialized job support.

Padata also supports multithreaded jobs, splitting up the job evenly while load balancing and coordinating between threads.

Running Serialized Jobs

Initializing

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

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

struct padata_instance *padata_alloc(const char *name);
```

'name' simply identifies the instance.

Then, complete padata initialization by allocating a padata_shell:

```
struct padata_shell *padata_alloc_shell(struct padata_instance *pinst);
```

A padata_shell is used to submit a job to padata and allows a series of such jobs to be serialized independently. A padata_instance may have one or more padata_shells associated with it, each allowing a separate series of jobs.

Modifying cpumasks

The CPUs used to run jobs can be changed in two ways, programatically with padata_set_cpumask() or via sysfs. The former is defined:

```
int padata_set_cpumask(struct padata_instance *pinst, int cpumask_type,
                       cpumask_var_t cpumask);
```

Here cpumask_type is one of PADATA_CPU_PARALLEL or PADATA_CPU_SERIAL, where a parallel cpumask describes which processors will be used to execute jobs submitted to this instance in parallel and a serial cpumask defines which processors are allowed to be used as the serialization callback processor. cpumask specifies the new cpumask to use.

There may be sysfs files for an instance's cpumasks. For example, pcrypt's live in /sys/kernel/pcrypt/<instance-name>. Within an instance's directory there are two files, parallel_cpumask and serial_cpumask, and either cpumask may be changed by echoing a bitmask into the file, for example:

```
echo f > /sys/kernel/pcrypt/pencrypt/parallel_cpumask
```

Reading one of these files shows the user-supplied cpumask, which may be different from the 'usable' cpumask.

Padata maintains two pairs of cpumasks internally, the user-supplied cpumasks and the 'usable' cpumasks. (Each pair consists of a parallel and a serial cpumask.) The user-supplied cpumasks default to all possible CPUs on instance allocation and may be changed as above. The usable cpumasks are always a subset of the user-supplied cpumasks and contain only the online CPUs in the user-supplied masks; these are the cpumasks padata actually uses. So it is legal to supply a cpumask to padata that contains offline CPUs. Once an offline CPU in the user-supplied cpumask comes online, padata is going to use it.

Changing the CPU masks are expensive operations, so it should not be done with great frequency.

Running A Job

Actually submitting work to the padata instance requires the creation of a padata_priv structure, which represents one job:

```
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 of its fields are private to padata, but the structure should be zeroed at initialisation 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 the job is done with:

```
int padata_do_parallel(struct padata_shell *ps,
                       struct padata_priv *padata, int *cb_cpu);
```

The ps and padata structures must be set up as described above; cb_cpu points to the preferred CPU to be used for the final callback when the job is done; it must be in the current instance's CPU mask (if not the cb_cpu pointer is updated to point to the CPU actually chosen). The return value from padata_do_parallel() is zero on success, indicating that the job is in progress. -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 the serial cpumask, no online CPUs in the parallel or serial cpumasks, or a stopped instance.

Each job 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 jobs. parallel() runs 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 job from this point. The job 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.

Serializing Jobs

When a job does complete, parallel() (or whatever function actually finishes the work) 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 run with local software interrupts disabled. Note that this call may be deferred for a while since the padata code takes pains to ensure that jobs are completed in the order in which they were submitted.

Destroying

Cleaning up a padata instance predictably involves calling the two free functions that correspond to the allocation in reverse:

```
void padata_free_shell(struct padata_shell *ps);
void padata_free(struct padata_instance *pinst);
```

It is the user's responsibility to ensure all outstanding jobs are complete before any of the above are called.

Running Multithreaded Jobs

A multithreaded job has a main thread and zero or more helper threads, with the main thread participating in the job and then waiting until all helpers have finished. padata splits the job into units called chunks, where a chunk is a piece of the job that one thread completes in one call to the thread function.

A user has to do three things to run a multithreaded job. First, describe the job by defining a padata_mt_job structure, which is explained in the Interface section. This includes a pointer to the thread function, which padata will call each time it assigns a job chunk to a thread. Then, define the thread function, which accepts three arguments, `start`, `end`, and `arg`, where the first two delimit the range that the thread operates on and the last is a pointer to the job's shared state, if any. Prepare the shared state, which is typically allocated on the main thread's stack. Last, call padata_do_multithreaded(), which will return once the job is finished.

Interface

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\core-api\[linux-master][Documentation][core-api]padata.rst, line 177)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: include/linux/padata.h
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\core-api\[linux-master][Documentation][core-api]padata.rst, line 178)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: kernel/padata.c
```