**What is a workqueue?**

There are many cases where asynchronous process execution is required and WQ is most commonly used mechanism.

**Old workqueue implementation:**

At highest level, there are worker threads.

* There can be multiple types of worker threads.
* There is one worker thread per processor of given type.
* Parts of kernel can create threads as needed.
* By default, there is events worker thread.
* Each worker thread is represented by cpu\_workqueue\_struct structure.
* Worqueue\_struct structure represents all worker threads of a given type.

An example,

Assumptions

* You created a *falcon* worker type.
* Assume you have 4 processor cores.

Then

* there are four event threads(and thus four cpu\_workqueue\_structs), and
* there are four *falcon* threads(and thus another four cpu\_workqueue\_structs)
* there is one workqueue\_struct for the *events*  and one for the *falcon* type.

Approaching from lowest level, which starts from work.

* Your driver creates work, which it want to defer.( work\_struct represents this work)
* Work is submitted to a specific worker thread, in our case a specific falcon thread.
* The worker then wakes up and performs the queued work.

Using Work Queues

Scheduling Work to default worker threads,

schedule\_work(&work);

schedule\_delayed\_work(&work, delay);

Creating new work queues

Struct workqueue\_struct \*create\_workqueue(const char \*name);

This function creates all the worker thread( one for each processor in the system)

Scheduling work on your private work\_queues,

queue\_work(wq, work)

**Drawbacks of old work queue implemtations:**

* One worker thread per cpu per workqueue.
* As Numbers of MT WQ users are increasing in kernel and with number of cores continuously rising, some system saturated the default 32K PID space just booting up.
* Workqueue threads contend with each other for the CPU, causing more context switches than are really necessary
* Although they waste lot of resources, the level of concurrency was unsatisfactory.
* A MT wq could only provide one execution context per CPU.
* Work items have to compete for those very limited execution contexts.
* As MT wq don’t provide much better concurrency, users had to implement their own work queue.

Concurrency Managed Work queues – The new implementation

Short term goals,

* Reduce the number of kernel threads running on the system. Provide single unified worker pool per cpu which can be shared by all users.
* Use what is necessary and allocate resources lazily on demand.
* While simultaneously increasing the concurrency of tasks submitted to workqueues.(you need to get back here for explanation)

per-workqueue kernel threads are gone, replaced by a central set of threads with names like [kworker/0:0]

There are no threads dedicated to any specific workqueue. Instead, there is a global pool of threads attached to each CPU in the system

When a work item is enqueued, it will be passed to one of the global threads at the right time (as deemed by the workqueue code).

There are two worker-pools, one for normal work items and the other

for high priority ones, for each possible CPU and some extra

worker-pools to serve work items queued on unbound workqueues - the

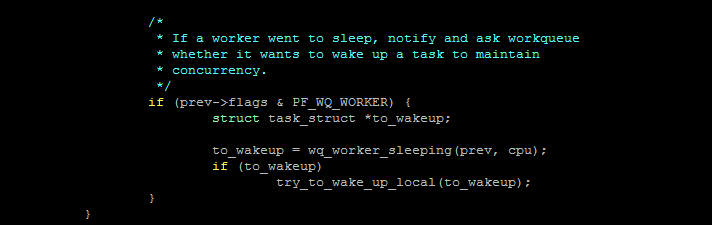
number of these backing pools is dynamic.

One interesting implication of this change is that tasks submitted to the same workqueue on the same CPU may now execute concurrently - something which does not happen with previous workqueues. This is because we have multiple kworker threads in thread pool compared to one thread per cpu in earlier case.

No of thread configurable?

Each worker pool bound to an actual cpu implements concurrency management by hooking in to the scheduler.

\_\_schedule() File:kernel/core/sched.c



This function tells the workqueue subsystem, that a worker is going to sleep, the worker in the same cpu can be woken up by returning pointer to its task.

wq\_worker\_sleeping function checks the concurrency level of the of the worker\_pool associated with this worker.

API

Alloc\_wokrqueue() allocates a wq.

3 args name, flags and max\_active.

Flags and max\_active how items are assigned execution resources, scheduled and executed.

Flags :

WQ\_UNBOUND

Workqueue latencies would be much lower with cmwq because works don't need to  
wait for other works on the same wq to complete. So, in general, the latencies will be more predictable and lower.

Alloc\_orderd\_worqueue

Freezable workqueues

Concurrency management doesn't apply to unbound ones tho.

Currently each workqueue has its own dedicated worker pool. This

causes the following problems.

\* Works which are dependent on each other can cause a deadlock by

depending on the same execution resource. This is bad because this

type of dependency is quite difficult to find.

\* Works which may sleep and take long time to finish need to have

separate workqueues so that it doesn't block other works. Similarly

works which want to be executed in timely manner often need to

create it custom workqueue too to avoid being blocked by long

running ones. This leads to large number of workqueues and thus

many workers.

\* The static one-per-cpu worker isn't good enough for jobs which

require higher level of concurrency necessiating other worker pool

mechanism. slow-work and async are good examples and there are also

some custom implementations buried in subsystems.

\* Combined, the above factors lead to many workqueues with large

number of dedicated and mostly unused workers. This also makes work

processing less optimal as the dedicated workers end up switching

among themselves costing scheduleing overhead and wasting cache

footprint for their stacks and as the system gets busy, these

workers end up competing with each other.

To solve the above issues, this patch implements concurrency-managed

workqueue.

There is single global cpu workqueue (gcwq) for each cpu which serves

all the workqueues. gcwq maintains single pool of workers which is

shared by all cwqs on the cpu.

gcwq keeps the number of concurrent active workers to minimum but no

less. As long as there's one or more running workers on the cpu, no

new worker is scheduled so that works can be processed in batch as

much as possible but when the last running worker blocks, gcwq

immediately schedules new worker so that the cpu doesn't sit idle

while there are works to be processed.

gcwq always keeps at least single idle worker around. When a new

worker is necessary and the worker is the last idle one, the worker

assumes the role of "manager" and manages the worker pool -

ie. creates another worker. Forward-progress is guaranteed by having

dedicated rescue workers for workqueues which may be necessary while

creating a new worker. When the manager is having problem creating a

new worker, mayday timer activates and rescue workers are summoned to

the cpu and execute works which may be necessary to create new

workers.

To keep track of which worker is executing which work, gcwq uses a

hash table. This is necessary as works may be destroyed once it

starts executing and flushing should be implemented by tracking

whether any worker is executing the work.

cpu hotplug implementation is more complex than before because there

are multiple workers and now workqueue is capable of hosting long

erunning works. cpu offlining is implemented by creating a "trustee"

kthread which runs the gcwq as if the cpu is still online until all

works are drained. As soon as the trustee takes over the gcwq, cpu

hotunplug operation can proceed without waiting for workqueues to be

drained. Onlining is the reverse. If trustee is still trying to

drain the gcwq from the previous offlining, it puts all workers back

to the cpu and let the gcwq run as if cpu has been online the whole

time.

The new implementation has the following benefits.

\* Workqueue users no longer have to worry about managing concurrency

and, in most cases, deadlocks. The workqueue will manage it

automatically and unless the deadlock chain involves more than 127

works, it won't happen.

\* There's one single shared pool of workers per cpu and one rescuer

for each workqueue which requires it, so there are far fewer number

of kthreads.

\* More efficient. Although it adds considerable amount of code, the

code added to hot path isn't big and works will be executed on the

local cpu and in batch as much as possible using minimal number of

kthreads leading to fewer task switches and lower cache

footprint. <NEED SOME BACKING NUMBERS>

\* As concurrency is no longer a problem, most types of asynchronous

jobs can be done using generic workqueue and other async mechanisms

can be removed.

sched: add hooks for workqueue

Concurrency managed workqueue needs to know when workers are going to

sleep and waking up. Using these two hooks, cmwq keeps track of the

current concurrency level and throttles execution of new works if it's

too high and wakes up another worker from the sleep hook if it becomes

too low.

This patch introduces PF\_WQ\_WORKER to identify workqueue workers and

adds the following two hooks.

\* wq\_worker\_waking\_up(): called when a worker is woken up.

\* wq\_worker\_sleeping(): called when a worker is going to sleep and may

return a pointer to a local task which should be woken up. The

returned task is woken up using try\_to\_wake\_up\_local() which is

simplified ttwu which is called under rq lock and can only wake up

local tasks.

Both hooks are currently defined as noop in kernel/workqueue\_sched.h.

Later cmwq implementation will replace them with proper

implementation.

These hooks are hard coded as they'll always be enabled.

===============  
On Wed, Mar 19, 2014 at 08:34:07PM +0100, Peter Zijlstra wrote:  
> The way I understand workqueues is that we cannot guarantee concurrency  
> like this. It tries, but there's no guarantee.  
  
So, the guarantee is that if a workqueue has WQ\_MEM\_RECLAIM, it'll  
always have at least one worker thread working on it, so workqueues  
which may be depended upon during memory reclaim should have the flag  
set and must not require more than single level of concurrency to make  
forward progress. Workqueues w/o memory reclaim set depend on the  
fact that eventually memory will be reclaimed and enough number of  
workers necessary to make forward progress will be made available.  
  
> WQ\_MAX\_ACTIVE seems to be a hard upper limit of concurrent workers. So  
> given 511 other blocked works, the described problem will always happen.  
  
That actually is per-workqueue limit and workqueue core will try to  
create as many workers as possible to satisfy the demanded  
concurrency. ie. having two workqueues with the same max\_active means  
that the total number of workers may reach 2 \* max\_active; however,  
this is no guarantee. If the system is under memory pressure and the  
workqueues don't have MEM\_RECLAIM set, they may not get any  
concurrency until more memory is made available.

This patch makes all workqueues non-reentrant. If a work item is

executing on a different CPU when queueing is requested, it is always

queued to that CPU. This guarantees that any given work item can be

executing on one CPU at maximum and if a work item is queued and

executing, both are on the same CPU.

The only behavior change which may affect workqueue users negatively

is that non-reentrancy overrides the affinity specified by

queue\_work\_on(). On a reentrant workqueue, the affinity specified by

queue\_work\_on() is always followed. Now, if the work item is

executing on one of the CPUs, the work item will be queued there

regardless of the requested affinity. I've reviewed all workqueue

users which request explicit affinity, and, fortunately, none seems to

be crazy enough to exploit parallel execution of the same work item.

This adds an additional busy\_hash lookup if the work item was

previously queued on a different CPU. This shouldn't be noticeable

under any sane workload. Work item queueing isn't a very

high-frequency operation and they don't jump across CPUs all the time.

In a micro benchmark to exaggerate this difference - measuring the

time it takes for two work items to repeatedly jump between two CPUs a

number (10M) of times with busy\_hash table densely populated, the

difference was around 3%.

While the overhead is measureable, it is only visible in pathological

cases and the difference isn't huge. This change brings much needed

sanity to workqueue and makes its behavior consistent with timer. I

think this is the right tradeoff to make.

This enables significant simplification of workqueue API.

Simplification patches will follow.