# Concurrency Managed Workqueue (cmwq)

September, 2010 Tejun Heo <tj@kernel.org>

Florian Mickler <florian@mickler.org>

CONTENTS

1. Introduction

2. Why cmwq?

3. The Design

4. Application Programming Interface (API)

5. Example Execution Scenarios

6. Guidelines

7. Debugging

# 1. Introduction

There are many cases where an asynchronous process execution context

is needed and the workqueue (wq) API is the most commonly used

mechanism for such cases.

When such an asynchronous execution context is needed, a work item

describing which function to execute is put on a queue. An

independent thread serves as the asynchronous execution context. The

queue is called workqueue and the thread is called worker.

While there are work items on the workqueue the worker executes the

functions associated with the work items one after the other. When

there is no work item left on the workqueue the worker becomes idle.

When a new work item gets queued, the worker begins executing again.

# 2. Why cmwq?

In the original wq implementation, a multi threaded (MT) wq had one

worker thread per CPU and a single threaded (ST) wq had one worker

thread system-wide. A single MT wq needed to keep around the same

number of workers as the number of CPUs. The kernel grew a lot of MT

wq users over the years and with the number of CPU cores continuously

rising, some systems saturated the default 32k PID space just booting

up.

Although MT wq wasted a lot of resource, the level of concurrency

provided was unsatisfactory. The limitation was common to both ST and

MT wq albeit less severe on MT. Each wq maintained its own separate

worker pool. A MT wq could provide only one execution context per CPU

while a ST wq one for the whole system. Work items had to compete for

those very limited execution contexts leading to various problems

including proneness to deadlocks around the single execution context.

The tension between the provided level of concurrency and resource

usage also forced its users to make unnecessary tradeoffs like libata

choosing to use ST wq for polling PIOs and accepting an unnecessary

limitation that no two polling PIOs can progress at the same time. As

MT wq don't provide much better concurrency, users which require

higher level of concurrency, like async or fscache, had to implement

their own thread pool.

Concurrency Managed Workqueue (cmwq) is a reimplementation of wq with

focus on the following goals.

\* Maintain compatibility with the original workqueue API.

\* Use per-CPU unified worker pools shared by all wq to provide

flexible level of concurrency on demand without wasting a lot of

resource.

\* Automatically regulate worker pool and level of concurrency so that

the API users don't need to worry about such details.

# 3. The Design

In order to ease the asynchronous execution of functions a new

abstraction, the work item, is introduced.

A work item is a simple struct that holds a pointer to the function

that is to be executed asynchronously. Whenever a driver or subsystem

wants a function to be executed asynchronously it has to set up a work

item pointing to that function and queue that work item on a

workqueue.

Special purpose threads, called worker threads, execute the functions

off of the queue, one after the other. If no work is queued, the

worker threads become idle. These worker threads are managed in so

called worker-pools.

The cmwq design differentiates between the user-facing workqueues that

subsystems and drivers queue work items on and the backend mechanism

which manages worker-pools and processes the queued work items.

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.

Subsystems and drivers can create and queue work items through special

workqueue API functions as they see fit. They can influence some

aspects of the way the work items are executed by setting flags on the

workqueue they are putting the work item on. These flags include

things like CPU locality, concurrency limits, priority and more. To

get a detailed overview refer to the API description of

alloc\_workqueue() below.

When a work item is queued to a workqueue, the target worker-pool is

determined according to the queue parameters and workqueue attributes

and appended on the shared worklist of the worker-pool. For example,

unless specifically overridden, a work item of a bound workqueue will

be queued on the worklist of either normal or highpri worker-pool that

is associated to the CPU the issuer is running on.

For any worker pool implementation, managing the concurrency level

(how many execution contexts are active) is an important issue. cmwq

tries to keep the concurrency at a minimal but sufficient level.

Minimal to save resources and sufficient in that the system is used at

its full capacity.

Each worker-pool bound to an actual CPU implements concurrency

management by hooking into the scheduler. The worker-pool is notified

whenever an active worker wakes up or sleeps and keeps track of the

number of the currently runnable workers. Generally, work items are

not expected to hog a CPU and consume many cycles. That means

maintaining just enough concurrency to prevent work processing from

stalling should be optimal. As long as there are one or more runnable

workers on the CPU, the worker-pool doesn't start execution of a new

work, but, when the last running worker goes to sleep, it immediately

schedules a new worker so that the CPU doesn't sit idle while there

are pending work items. This allows using a minimal number of workers

without losing execution bandwidth.

Keeping idle workers around doesn't cost other than the memory space

for kthreads, so cmwq holds onto idle ones for a while before killing

them.

For unbound workqueues, the number of backing pools is dynamic.

Unbound workqueue can be assigned custom attributes using

apply\_workqueue\_attrs() and workqueue will automatically create

backing worker pools matching the attributes. The responsibility of

regulating concurrency level is on the users. There is also a flag to

mark a bound wq to ignore the concurrency management. Please refer to

the API section for details.

Forward progress guarantee relies on that workers can be created when

more execution contexts are necessary, which in turn is guaranteed

through the use of rescue workers. All work items which might be used

on code paths that handle memory reclaim are required to be queued on

wq's that have a rescue-worker reserved for execution under memory

pressure. Else it is possible that the worker-pool deadlocks waiting

for execution contexts to free up.

# 4. Application Programming Interface (API)

alloc\_workqueue() allocates a wq. The original create\_\*workqueue()

functions are deprecated and scheduled for removal. alloc\_workqueue()

takes three arguments - @name, @flags and @max\_active. @name is the

name of the wq and also used as the name of the rescuer thread if

there is one.

A wq no longer manages execution resources but serves as a domain for

forward progress guarantee, flush and work item attributes. @flags

and @max\_active control how work items are assigned execution

resources, scheduled and executed.

@flags:

WQ\_UNBOUND

Work items queued to an unbound wq are served by the special

woker-pools which host workers which are not bound to any

specific CPU. This makes the wq behave as a simple execution

context provider without concurrency management. The unbound

worker-pools try to start execution of work items as soon as

possible. Unbound wq sacrifices locality but is useful for

the following cases.

\* Wide fluctuation in the concurrency level requirement is

expected and using bound wq may end up creating large number

of mostly unused workers across different CPUs as the issuer

hops through different CPUs.

\* Long running CPU intensive workloads which can be better

managed by the system scheduler.

WQ\_FREEZABLE

A freezable wq participates in the freeze phase of the system

suspend operations. Work items on the wq are drained and no

new work item starts execution until thawed.

WQ\_MEM\_RECLAIM

All wq which might be used in the memory reclaim paths \_MUST\_

have this flag set. The wq is guaranteed to have at least one

execution context regardless of memory pressure.

WQ\_HIGHPRI

Work items of a highpri wq are queued to the highpri

worker-pool of the target cpu. Highpri worker-pools are

served by worker threads with elevated nice level.

Note that normal and highpri worker-pools don't interact with

each other. Each maintain its separate pool of workers and

implements concurrency management among its workers.

WQ\_CPU\_INTENSIVE

Work items of a CPU intensive wq do not contribute to the

concurrency level. In other words, runnable CPU intensive

work items will not prevent other work items in the same

worker-pool from starting execution. This is useful for bound

work items which are expected to hog CPU cycles so that their

execution is regulated by the system scheduler.

Although CPU intensive work items don't contribute to the

concurrency level, start of their executions is still

regulated by the concurrency management and runnable

non-CPU-intensive work items can delay execution of CPU

intensive work items.

This flag is meaningless for unbound wq.

Note that the flag WQ\_NON\_REENTRANT no longer exists as all workqueues

are now non-reentrant - any work item is guaranteed to be executed by

at most one worker system-wide at any given time.

@max\_active:

@max\_active determines the maximum number of execution contexts per

CPU which can be assigned to the work items of a wq. For example,

with @max\_active of 16, at most 16 work items of the wq can be

executing at the same time per CPU.

Currently, for a bound wq, the maximum limit for @max\_active is 512

and the default value used when 0 is specified is 256. For an unbound

wq, the limit is higher of 512 and 4 \* num\_possible\_cpus(). These

values are chosen sufficiently high such that they are not the

limiting factor while providing protection in runaway cases.

The number of active work items of a wq is usually regulated by the

users of the wq, more specifically, by how many work items the users

may queue at the same time. Unless there is a specific need for

throttling the number of active work items, specifying '0' is

recommended.

Some users depend on the strict execution ordering of ST wq. The

combination of @max\_active of 1 and WQ\_UNBOUND is used to achieve this

behavior. Work items on such wq are always queued to the unbound

worker-pools and only one work item can be active at any given time thus

achieving the same ordering property as ST wq.

# 5. Example Execution Scenarios

The following example execution scenarios try to illustrate how cmwq

behave under different configurations.

Work items w0, w1, w2 are queued to a bound wq q0 on the same CPU.

w0 burns CPU for 5ms then sleeps for 10ms then burns CPU for 5ms

again before finishing. w1 and w2 burn CPU for 5ms then sleep for

10ms.

Ignoring all other tasks, works and processing overhead, and assuming

simple FIFO scheduling, the following is one highly simplified version

of possible sequences of events with the original wq.

TIME IN MSECS EVENT

0 w0 starts and burns CPU

5 w0 sleeps

15 w0 wakes up and burns CPU

20 w0 finishes

20 w1 starts and burns CPU

25 w1 sleeps

35 w1 wakes up and finishes

35 w2 starts and burns CPU

40 w2 sleeps

50 w2 wakes up and finishes

And with cmwq with @max\_active >= 3,

TIME IN MSECS EVENT

0 w0 starts and burns CPU

5 w0 sleeps

5 w1 starts and burns CPU

10 w1 sleeps

10 w2 starts and burns CPU

15 w2 sleeps

15 w0 wakes up and burns CPU

20 w0 finishes

20 w1 wakes up and finishes

25 w2 wakes up and finishes

If @max\_active == 2,

TIME IN MSECS EVENT

0 w0 starts and burns CPU

5 w0 sleeps

5 w1 starts and burns CPU

10 w1 sleeps

15 w0 wakes up and burns CPU

20 w0 finishes

20 w1 wakes up and finishes

20 w2 starts and burns CPU

25 w2 sleeps

35 w2 wakes up and finishes

Now, let's assume w1 and w2 are queued to a different wq q1 which has

WQ\_CPU\_INTENSIVE set,

TIME IN MSECS EVENT

0 w0 starts and burns CPU

5 w0 sleeps

5 w1 and w2 start and burn CPU

10 w1 sleeps

15 w2 sleeps

15 w0 wakes up and burns CPU

20 w0 finishes

20 w1 wakes up and finishes

25 w2 wakes up and finishes

6. Guidelines

\* Do not forget to use WQ\_MEM\_RECLAIM if a wq may process work items

which are used during memory reclaim. Each wq with WQ\_MEM\_RECLAIM

set has an execution context reserved for it. If there is

dependency among multiple work items used during memory reclaim,

they should be queued to separate wq each with WQ\_MEM\_RECLAIM.

\* Unless strict ordering is required, there is no need to use ST wq.

\* Unless there is a specific need, using 0 for @max\_active is

recommended. In most use cases, concurrency level usually stays

well under the default limit.

\* A wq serves as a domain for forward progress guarantee

(WQ\_MEM\_RECLAIM, flush and work item attributes. Work items which

are not involved in memory reclaim and don't need to be flushed as a

part of a group of work items, and don't require any special

attribute, can use one of the system wq. There is no difference in

execution characteristics between using a dedicated wq and a system

wq.

\* Unless work items are expected to consume a huge amount of CPU

cycles, using a bound wq is usually beneficial due to the increased

level of locality in wq operations and work item execution.

7. Debugging

Because the work functions are executed by generic worker threads

there are a few tricks needed to shed some light on misbehaving

workqueue users.

Worker threads show up in the process list as:

root 5671 0.0 0.0 0 0 ? S 12:07 0:00 [kworker/0:1]

root 5672 0.0 0.0 0 0 ? S 12:07 0:00 [kworker/1:2]

root 5673 0.0 0.0 0 0 ? S 12:12 0:00 [kworker/0:0]

root 5674 0.0 0.0 0 0 ? S 12:13 0:00 [kworker/1:0]

If kworkers are going crazy (using too much cpu), there are two types

of possible problems:

1. Something being scheduled in rapid succession

2. A single work item that consumes lots of cpu cycles

The first one can be tracked using tracing:

$ echo workqueue:workqueue\_queue\_work > /sys/kernel/debug/tracing/set\_event

$ cat /sys/kernel/debug/tracing/trace\_pipe > out.txt

(wait a few secs)

^C

If something is busy looping on work queueing, it would be dominating

the output and the offender can be determined with the work item

function.

For the second type of problems it should be possible to just check

the stack trace of the offending worker thread.

$ cat /proc/THE\_OFFENDING\_KWORKER/stack

The work item's function should be trivially visible in the stack

trace.