Apache HTTP Server Version 2.4

Apache MPM event

Description:	A variant of the `worker` MPM with the goal of consuming threads only for connections with active processing
Status:	MPM
Module Identifier:	mpm_event_module
Source File:	event.c

Summary

The `event` Multi-Processing Module (MPM) is designed to allow more requests to be served simultaneously by passing off some processing work to the listeners threads, freeing up the worker threads to serve new requests.

To use the `event` MPM, add `--with-mpm=event` to the `configure` script's arguments when building the `httpd`.

Topics

Directives

Bugfix checklist

- httpd changelog
- Known issues
- Report a bug

See also

- The worker MPM
- Comments

Relationship with the Worker MPM

`event` is based on the `worker` MPM, which implements a hybrid multi-process multi-threaded server. A single control process (the parent) is responsible for launching child processes. Each child process creates a fixed number of server threads as specified in the `ThreadsPerChild` directive, as well as a listener thread which listens for connections and passes them to a worker thread for processing when they arrive.

Run-time configuration directives are identical to those provided by `worker`, with the only addition of the `AsyncRequestWorkerFactor`.

How it Works

This MPM tries to fix the 'keep alive problem' in HTTP. After a client completes the first request, it can keep the connection open, sending further requests using the same socket and saving significant overhead in creating TCP connections. However, Apache HTTP Server traditionally keeps an entire child process/thread waiting for data from the client, which brings its own disadvantages. To solve this problem, this MPM uses a dedicated listener thread for each process to handle both the Listening sockets, all sockets that are in a Keep Alive state, sockets where the handler and protocol filters have done their work and the ones where the only remaining thing to do is send the data to the client.

This new architecture, leveraging non-blocking sockets and modern kernel features exposed by APR (↗ ../glossary.html#apr) (like Linux's epoll), no longer requires the `mpm-accept` `Mutex` configured to avoid the thundering herd problem.

The total amount of connections that a single process/threads block can handle is regulated by the `AsyncRequestWorkerFactor` directive.

Async connections

Async connections would need a fixed dedicated worker thread with the previous MPMs but not with event. The status page of `mod_status` shows new columns under the Async connections section:

Writing
: While sending the response to the client, it might happen that the TCP write buffer fills up because the connection is too slow. Usually in this case a `write()` to the socket returns `EWOULDBLOCK` or `EAGAIN`, to become writable again after an idle time. The worker holding the socket might be able to offload the waiting task to the listener thread, that in turn will re-assign it to the first idle worker thread available once an event will be raised for the socket (for example, "the socket is now writable"). Please check the Limitations section for more information.

Keep-alive
: Keep Alive handling is the most basic improvement from the worker MPM. Once a worker thread finishes to flush the response to the client, it can offload the socket handling to the listener thread, that in turns will wait for any event from the OS, like "the socket is readable". If any new request comes from the client, then the listener will forward it to the first worker thread available. Conversely, if the `KeepAliveTimeout` occurs

then the socket will be closed by the listener. In this way the worker threads are not responsible for idle sockets and they can be re-used to serve other requests.

Closing
Sometimes the MPM needs to perform a lingering close, namely sending back an early error to the client while it is still transmitting data to httpd. Sending the response and then closing the connection immediately is not the correct thing to do since the client (still trying to send the rest of the request) would get a connection reset and could not read the httpd's response. The lingering close is time bounded but it can take relatively long time, so it's offloaded to a worker thread (including the shutdown hooks and real socket close). From 2.4.28 onward this is also the case when connections finally timeout (the listener thread never handles connections besides waiting for and dispatching their events).

These improvements are valid for both HTTP/HTTPS connections.

Graceful process termination and Scoreboard usage

This mpm showed some scalability bottlenecks in the past leading to the following error: "**scoreboard is full, not at MaxRequestWorkers**". `MaxRequestWorkers` limits the number of simultaneous requests that will be served at any given time and also the number of allowed processes (`MaxRequestWorkers` / `ThreadsPerChild`), meanwhile the Scoreboard is a representation of all the running processes and the status of their worker threads. If the scoreboard is full (so all the threads have a state that is not idle) but the number of active requests served is not `MaxRequestWorkers`, it means that some of them are blocking new requests that could be served but that are queued instead (up to the limit imposed by `ListenBacklog`). Most of the times the threads are stuck in the Graceful state, namely they are waiting to finish their work with a TCP connection to safely terminate and free up a scoreboard slot (for example handling long running requests, slow clients or connections with keep-alive enabled). Two scenarios are very common:

- During a graceful restart. The parent process signals all its children to complete their work and terminate, while it reloads the config and forks new processes. If the old children keep running for a while before stopping, the scoreboard will be partially occupied until their slots are freed.
- When the server load goes down in a way that causes httpd to stop some processes (for example due to `MaxSpareThreads`). This is particularly problematic because when the load increases again, httpd will try to start new processes. If the pattern repeats, the number of processes can rise quite a bit, ending up in a mixture of old processes trying to stop and new ones trying to do some work.

From 2.4.24 onward, mpm-event is smarter and it is able to handle graceful terminations in a much better way. Some of the improvements are:

- Allow the use of all the scoreboard slots up to `ServerLimit`. `MaxRequestWorkers` and `ThreadsPerChild` are used to limit the amount of active processes, meanwhile `ServerLimit` takes also into account the ones doing a graceful close to allow extra slots when needed. The idea is to use `ServerLimit` to instruct httpd about how many overall processes are tolerated before impacting the system resources.
- Force gracefully finishing processes to close their connections in keep-alive state.
- During graceful shutdown, if there are more running worker threads than open connections for a given process, terminate these threads to free resources faster (which may be needed for new processes).
- If the scoreboard is full, prevent more processes to finish gracefully due to reduced load until old processes have terminated (otherwise the situation would get worse once the load increases again).

The behavior described in the last point is completely observable via `mod_status` in the connection summary table through two new columns: "Slot" and "Stopping". The former indicates the PID and the latter if the process is stopping or not; the extra state "Yes (old gen)" indicates a process still running after a graceful restart.

Limitations

The improved connection handling may not work for certain connection filters that have declared themselves as incompatible with event. In these cases, this MPM will fall back to the behavior of the `worker` MPM and reserve

one worker thread per connection. All modules shipped with the server are compatible with the event MPM.

A similar restriction is currently present for requests involving an output filter that needs to read and/or modify the whole response body. If the connection to the client blocks while the filter is processing the data, and the amount of data produced by the filter is too big to be buffered in memory, the thread used for the request is not freed while httpd waits until the pending data is sent to the client.
To illustrate this point we can think about the following two situations: serving a static asset (like a CSS file) versus serving content retrieved from FCGI/CGI or a proxied server. The former is predictable, namely the event MPM has full visibility on the end of the content and it can use events: the worker thread serving the response content can flush the first bytes until `EWOULDBLOCK` or `EAGAIN` is returned, delegating the rest to the listener. This one in turn waits for an event on the socket, and delegates the work to flush the rest of the content to the first idle worker thread. Meanwhile in the latter example (FCGI/CGI/proxied content) the MPM can't predict the end of the response and a worker thread has to finish its work before returning the control to the listener. The only alternative is to buffer the response in memory, but it wouldn't be the safest option for the sake of the server's stability and memory footprint.

Background material

The event model was made possible by the introduction of new APIs into the supported operating systems:

- epoll (Linux)
- kqueue (BSD)
- event ports (Solaris)

Before these new APIs where made available, the traditional `select` and `poll` APIs had to be used. Those APIs get slow if used to handle many connections or if the set of connections rate of change is high. The new APIs allow to monitor much more connections and they perform way better when the set of connections to monitor changes frequently. So these APIs made it possible to write the event MPM, that scales much better with the typical HTTP pattern of many idle connections.

The MPM assumes that the underlying `apr_pollset` implementation is reasonably threadsafe. This enables the MPM to avoid excessive high level locking, or having to wake up the listener thread in order to send it a keep-alive socket. This is currently only compatible with KQueue and EPoll.

Requirements

This MPM depends on APR (↗ ../glossary.html#apr) 's atomic compare-and-swap operations for thread synchronization. If you are compiling for an x86 target and you don't need to support 386s, or you are compiling for a SPARC and you don't need to run on pre-UltraSPARC chips, add `--enable-nonportable-atomics=yes` to the `configure` script's arguments. This will cause APR to implement atomic operations using efficient opcodes not available in older CPUs.

This MPM does not perform well on older platforms which lack good threading, but the requirement for EPoll or KQueue makes this moot.

- To use this MPM on FreeBSD, FreeBSD 5.3 or higher is recommended. However, it is possible to run this MPM on FreeBSD 5.2.1, if you use `libkse` (see `man libmap.conf`).
- For NetBSD, at least version 2.0 is recommended.
- For Linux, a 2.6 kernel is recommended. It is also necessary to ensure that your version of `glibc` has been compiled with support for EPoll.

AsyncRequestWorkerFactor Directive

Description:	Limit concurrent connections per process
Syntax:	`AsyncRequestWorkerFactor` *factor*
Default:	`2`

Context:	server config
Status:	MPM
Module:	event
Compatibility:	Available in version 2.3.13 and later

The event MPM handles some connections in an asynchronous way, where request worker threads are only allocated for short periods of time as needed, and other connections with one request worker thread reserved per connection. This can lead to situations where all workers are tied up and no worker thread is available to handle new work on established async connections.

To mitigate this problem, the event MPM does two things:

- it limits the number of connections accepted per process, depending on the number of idle request workers;
- if all workers are busy, it will close connections in keep-alive state even if the keep-alive timeout has not expired. This allows the respective clients to reconnect to a different process which may still have worker threads available.

This directive can be used to fine-tune the per-process connection limit. A **process** will only accept new connections if the current number of connections (not counting connections in the "closing" state) is lower than:

`ThreadsPerChild` + (`AsyncRequestWorkerFactor` * *number of idle workers*)

An estimation of the maximum concurrent connections across all the processes given an average value of idle worker threads can be calculated with:

(`ThreadsPerChild` + (`AsyncRequestWorkerFactor` * *number of idle workers*)) * `ServerLimit`

Example

```
ThreadsPerChild = 10
ServerLimit = 4
AsyncRequestWorkerFactor = 2
MaxRequestWorkers = 40

idle_workers = 4 (average for all the processes to keep it simple)

max_connections = (ThreadsPerChild + (AsyncRequestWorkerFactor * idle_worke
                = (10 + (2 * 4)) * 4 = 72
```

When all the worker threads are idle, then absolute maximum numbers of concurrent connections can be calculared in a simpler way:

(`AsyncRequestWorkerFactor` + 1) * `MaxRequestWorkers`

Example

```
ThreadsPerChild = 10
ServerLimit = 4
MaxRequestWorkers = 40
AsyncRequestWorkerFactor = 2
```

If all the processes have all threads idle then:

```
idle_workers = 10
```

We can calculate the absolute maximum numbers of concurrent connections in two ways:

```
max_connections = (ThreadsPerChild + (AsyncRequestWorkerFactor * idle_worke
                = (10 + (2 * 10)) * 4 = 120

max_connections = (AsyncRequestWorkerFactor + 1) * MaxRequestWorkers
                = (2 + 1) * 40 = 120
```

Tuning `AsyncRequestWorkerFactor` requires knowledge about the traffic handled by httpd in each specific use case, so changing the default value requires extensive testing and data gathering from `mod_status`.

`MaxRequestWorkers` was called `MaxClients` prior to version 2.3.13. The above value shows that the old name did not accurately describe its meaning for the event MPM.

`AsyncRequestWorkerFactor` can take non-integer arguments, e.g "1.5".

Comments

Notice:
This is not a Q&A section. Comments placed here should be pointed towards suggestions on improving the documentation or server, and may be removed again by our moderators if they are either implemented or considered invalid/off-topic. Questions on how to manage the Apache HTTP Server should be directed at either our IRC channel, #httpd, on Freenode, or sent to our mailing lists.