ProcessesAndThreading

How processes and threads are used.

Updated Apr 17, 2011 by Graham.Dumpleton@gmail.com

If you are using mod_wsgi, please consider making a donation.

Processes And Threading

Apache can operate in a number of different modes dependent on the platform being used and the way in which it is configured. This ranges from multiple processes being used, with only one request being handled at a time within each process, to one or more processes being used, with concurrent requests being handled in distinct threads executing within those processes.

The combinations possible are further increased by mod_wsgi through its ability to create groups of daemon processes to which WSGI applications can be delegated. As with Apache itself, each process group can consist of one or more processes and optionally make use of multithreading. Unlike Apache, where some combinations are only possible based on how Apache was compiled, the mod_wsgi daemon processes can operate in any mode based only on runtime configuration settings.

This article provides background information on how Apache and mod_wsgi makes use of processes and threads to handle requests, and how Python sub interpreters are used to isolate WSGI applications. The implications of the various modes of operation on data sharing is also discussed.

WSGI Process/Thread Flags

Although Apache can make use of a combination of processes and/or threads to handle requests, this is not unique to the Apache web server and the WSGI specification acknowledges this fact. This acknowledgement is in the form of specific key/value pairs which must be supplied as part of the WSGI environment to a WSGI application. The purpose of these key/value pairs is to indicate whether the underlying web server does or does not make use of multiple processes and/or multiple threads to handle requests.

These key/value pairs are defined as follows in the WSGI specification.

wsgi.multithread

> This value should evaluate true if the application object may be simultaneously invoked by another thread in the same process, and should evaluate false otherwise.

wsgi.multiprocess

> This value should evaluate true if an equivalent application object may be simultaneously invoked by another process, and should evaluate false otherwise.

A WSGI application which is not written to take into consideration the different combinations of process and threading models may not be portable and potentially may not be robust when deployed to an alternate hosting platform or configuration.

Although you may not need an application or application component to work under all possible combinations for these values initially, it is highly recommended that any application component still be designed to work under any of the different operating modes. If for some reason this cannot be done due to the very nature of what functionality the component provides, the component should validate if it is being run within a compatible configuration and return a HTTP 500 internal server error response if it isn't.

An example of a component for which restrictions would apply is one providing an interactive browser based debugger session in response to an internal failure of a WSGI application. In this scenario, for the component to work correctly, subsequent HTTP requests must be processed by the same process. As such, the component can only be used with a web server that uses a single process. In other words, the value of 'wsgi.multiprocess' would have to evaluate to be false.

Multi-Processing Modules

The main factor which determines how Apache operates is which multi-processing module (MPM) is built into Apache at compile time. Although runtime configuration can customise the behaviour of the MPM, the choice of MPM will dictate whether or not multithreading is available.

On UNIX based systems, Apache defaults to being built with the 'prefork' MPM. If Apache 1.3 is being used this is actually the only choice, but for later versions of Apache, this can be overridden at build time by supplying an appropriate value in conjunction with the '--with-mpm' option when running the 'configure' script for Apache. The main alternative to the 'prefork' MPM which can be used on UNIX systems is the 'worker' MPM.

If you are unsure which MPM is built into Apache, it can be determined by running the Apache web server executable with the '-V' option. The output from running the web server executable with this option will be information about how it was configured when built.

```
Server version: Apache/2.2.1
Server built:   Mar  4 2007 20:48:15
Server's Module Magic Number: 20051115:1
Server loaded:  APR 1.2.6, APR-Util 1.2.6
```

```
Compiled using: APR 1.2.6, APR-Util 1.2.6
Architecture:   32-bit
Server MPM:     Worker
  threaded:     yes (fixed thread count)
    forked:     yes (variable process count)
Server compiled with....
 -D APACHE_MPM_DIR="server/mpm/worker"
 -D APR_HAS_MMAP
 -D APR_HAVE_IPV6 (IPv4-mapped addresses enabled)
 -D APR_USE_SYSVSEM_SERIALIZE
 -D APR_USE_PTHREAD_SERIALIZE
 -D SINGLE_LISTEN_UNSERIALIZED_ACCEPT
 -D APR_HAS_OTHER_CHILD
 -D AP_HAVE_RELIABLE_PIPED_LOGS
 -D DYNAMIC_MODULE_LIMIT=128
 -D HTTPD_ROOT="/usr/local/apache-2.2"
 -D SUEXEC_BIN="/usr/local/apache-2.2/bin/suexec"
 -D DEFAULT_SCOREBOARD="logs/apache_runtime_status"
 -D DEFAULT_ERRORLOG="logs/error_log"
 -D AP_TYPES_CONFIG_FILE="conf/mime.types"
 -D SERVER_CONFIG_FILE="conf/httpd.conf"
```

Which MPM is being used can be determined from the 'Server MPM' field.

On the Windows platform the only available MPM is 'winnt'.

The UNIX 'prefork' MPM

This MPM is the most commonly used. It was the only mode of operation available in Apache 1.3 and is still the default mode on UNIX systems in later versions of Apache. In this configuration, the main Apache process will at startup create multiple child processes. When a request is received by the parent process, it will be processed by which ever of the child processes is ready.

Each child process will only handle one request at a time. If another request arrives at the same time, it will be handled by the next available child process. When it is detected that the number of available processes is running out, additional child processes will be created as necessary. If a limit is specified as to the number of child processes which may be created and the limit is reached, plus there are sufficient requests arriving to fill up the listener socket queue, the client may instead receive an error resulting from not being able to establish a connection with the web server.

Where additional child processes have to be created due to a peak in the number of current requests arriving and where the number of requests has subsequently dropped off, the excess child processes may be shutdown and killed off. Child processes may also be shutdown and killed off after they have handled some set number of requests.

Although threads are not used to service individual requests, this does not preclude an application from creating separate threads to perform some specific task.

For the typical 'prefork' configuration where multiple processes are used, the WSGI environment key/value pairs indicating how processes and threads are being used will be as follows.

wsgi.multithread	False
wsgi.multiprocess	True

Because multiple processes are being used, a WSGI middleware component such as the interactive browser based debugger described would not be able to be used. If during development and testing of a WSGI application, use of such a debugger was required, the only option which would exist would be to limit the number of processes being used. This could be achieved using the Apache configuration:

```
StartServers 1
ServerLimit 1
```

With this configuration, only one process will be started, with no additional processes ever being created. The WSGI environment key/value pairs indicating how processes and threads are being used will for this configuration be as follows.

wsgi.multithread	False
wsgi.multiprocess	False

In effect, this configuration has the result of serialising all requests through a single process. This will allow an interactive browser based debugger to be used, but may prevent more complex WSGI applications which make use of AJAX techniques from working. This could occur where a web page initiates a sequence of AJAX requests and expects later requests to be able to complete while a response for an initial request is still pending. In other words, problems may occur where requests overlap, as subsequent requests will not be able to be executed until the initial request has completed.

The UNIX 'worker' MPM

The 'worker' MPM is similar to 'prefork' mode except that within each child process there will exist a number of worker threads. Instead of a request only being able to be processed by the next available idle child process and with the handling of the request being the only thing the child process is then doing, the request may be processed by a worker thread within a child process which already has other worker threads handling other requests at the same time.

It is possible that a WSGI application could be executed at the same time from multiple worker threads within the one child process. This means that multiple worker threads may want to access common shared data at the same time. As a consequence, such common shared data must be protected in a way that will allow access and modification in a thread safe manner. Normally this would necessitate the use of some form of synchronisation mechanism to ensure that only one thread at a time accesses and or modifies the common shared data.

If all worker threads within a child process were busy when a new request arrives the request would be processed by an idle worker thread in another child process. Apache may still create new child processes on demand if necessary. Apache may also still shutdown and kill off excess child processes, or child processes that have handled more than a set number of requests.

Overall, use of 'worker' MPM will result in less child processes needing to be created, but resource usage of individual child processes will be greater. On modern computer systems, the 'worker' MPM would in general be the prefered MPM to use and should if possible be used in preference to the 'prefork' MPM.

Although contention for the global interpreter lock (GIL) in Python can causes issues for pure Python programs, it is not generally as big an issue when using Python within Apache. This is because all the underlying infrastructure for accepting requests and mapping the URL to a WSGI application, as well as the handling of requests against static files are all performed by Apache in C code. While this code is being executed the thread will not be holding the Python GIL, thus allowing a greater level of overlapping execution where a system has multiple CPUs or CPUs with multiple cores.

This ability to make good use of more than processor, even when using multithreading, is further enchanced by the fact that Apache uses multiple processes for handling requests and not just a single process. Thus, even when there is some contention for the GIL within a specific process, it doesn't stop other processes from being able to run as the GIL is only local to a process and does not extend across processes.

For the typical 'worker' configuration where multiple processes and multiple threads are used, the WSGI environment key/value pairs indicating how processes and threads are being used will be as follows.

wsgi.multithread	True
wsgi.multiprocess	True

Similar to the 'prefork' MPM, the number of processes can be restricted to just one if required using the configuration:

```
StartServers 1
ServerLimit 1
```

With this configuration, only one process will be started, with no additional processes ever being created, but that one process would still make use of multiple threads.

The WSGI environment key/value pairs indicating how processes and threads are being used will for this configuration be as follows.

wsgi.multithread	True
wsgi.multiprocess	False

Because multiple threads are being used, there would be no problem with overlapping requests generated by an AJAX based web page.

The Windows 'winnt' MPM

On the Windows platform the 'winnt' MPM is the only option available. With this MPM, multiple worker threads within a child process are used to handle all requests. The 'winnt' MPM is different to the 'worker' mode however in that there is only one child process. At no time are additional child processes created, or that one child process shutdown and killed off, except where Apache as a whole is being stopped or restarted. Because there is only one child process, the maximum number of threads used is much greater.

The WSGI environment key/value pairs indicating how processes and threads are being used will for this configuration be as follows.

wsgi.multithread	True
wsgi.multiprocess	False

The mod_wsgi Daemon Processes

When using 'daemon' mode of mod_wsgi, each process group can be individually configured so as to run in a manner similar to either 'prefork', 'worker' or 'winnt' MPMs for Apache. This is achieved by controlling the number of processes and threads within each process using the 'processes' and 'threads' options of the WSGIDaemonProcess directive.

To emulate the same process/thread model as the 'winnt' MPM, that is, a single process with multiple threads, the following configuration would be used:

```
WSGIDaemonProcess example threads=25
```

The WSGI environment key/value pairs indicating how processes and threads are being used will for this configuration be as follows.

wsgi.multithread	True
wsgi.multiprocess	False

Note that by not specifying the 'processes' option only a single process is created within the process group. Although providing 'processes=1' as an option would also result in a single process being created, this has a slightly different meaning and so you should only do this if necessary.

The difference between not specifying the 'processes' option and defining 'processes=1' will be that WSGI environment attribute called 'wsgi.multiprocess' will be set to be True when the 'processes' option is defined, whereas not providing the option at all will result in the attribute being set to be False. This distinction is to allow for where some form of mapping mechanism might be used to distribute requests across multiple process groups and thus in effect it is still a multiprocess application.

In other words, if you use the configuration:

```
WSGIDaemonProcess example processes=1 threads=25
```

the WSGI environment key/value pairs indicating how processes and threads are being used will instead be:

wsgi.multithread	True
wsgi.multiprocess	True

If you need to ensure that 'wsgi.multiprocess' is False so that interactive debuggers do not complain about an incompatible configuration, simply do not specify the 'processes' option and allow the default behaviour of a single daemon process to apply.

To emulate the same process/thread model as the 'worker' MPM, that is, multiple processes with multiple threads, the following configuration would be used:

```
WSGIDaemonProcess example processes=2 threads=25
```

The WSGI environment key/value pairs indicating how processes and threads are being used will for this configuration be as follows.

wsgi.multithread	True
wsgi.multiprocess	True

To emulate the same process/thread model as the 'prefork' MPM, that is, multiple processes with only a single thread running in each, the following configuration would be used:

```
WSGIDaemonProcess example processes=5 threads=1
```

The WSGI environment key/value pairs indicating how processes and threads are being used will for this configuration be as follows.

wsgi.multithread	False
wsgi.multiprocess	True

Note that when using mod_wsgi daemon processes, the processes are only used to execute the Python based WSGI application. The processes are not in any way used to serve static files, or host applications implemented in other languages.

Unlike the normal Apache child processes when 'embedded' mode of mod_wsgi is used, the configuration as to the number of daemon processes within a process group is fixed. That is, when the server experiences additional load, no more daemon processes are created than what is defined. You should therefore always plan ahead and make sure the number of processes and threads defined is adequate to cope with the expected load.

Sharing Of Global Data

When the 'winnt' MPM is being used, or the 'prefork' or 'worker' MPM are forced to run with only a single process, all request handlers within a specific WSGI application will always be accessing the same global data. This global data will persist in memory until Apache is shutdown or restarted, or in the case of the 'prefork' or 'worker' MPM until the child process is recycled due to reaching a predefined request limit.

This ability to access the same global data and for that data to persist for the lifetime of the child process is not present when either of the 'prefork' or 'worker' MPM are used in multiprocess mode. In other words, where the WSGI environment key/value pair indicating how processes are used is set to:

wsgi.multiprocess	True

This is because request handlers can execute within the context of distinct child processes, each with their own set of global data unique to that child process.

The consequences of this are that you cannot assume that separate invocations of a request handler will have access to the same global data if that data only resides within the memory of the child process. If some set of global data must be accessible by all invocations of a handler, that data will need to be stored in a way that it can be accessed from multiple child processes. Such sharing could be achieved by storing the global data within an external database, the filesystem or in shared memory accessible by all child processes.

Since the global data will be accessible from multiple child processes at the same time, there must be adequate locking mechanisms in place to prevent distinct child processes from trying to modify the same data at the same time. The locking mechanisms need to also be able to deal with the case of multiple threads within one child process accessing the global data at the same time, as will be the case for the 'worker' and 'winnt' MPM.

Python Sub Interpreters

The default behaviour of mod_wsgi is to create a distinct Python sub interpreter for each WSGI application. Thus, where Apache is being used to host multiple WSGI applications a process will contain multiple sub interpreters. When Apache is run in a mode whereby there are multiple child processes, each child process will contain sub interpreters for each WSGI application.

When a sub interpreter is created for a WSGI application, it would then normally persist for the life of the process. The only exception to this would be where interpreter reloading is enabled, in which case the sub interpreter would be destroyed and recreated when the WSGI application script file has been changed.

For the sub interpreter created for each WSGI application, they will each have their own set of Python modules. In other words, a change to the global data within the context of one sub interpreter will not be seen from the sub interpreter corresponding to a different WSGI application. This will be the case whether or not the sub interpreters are in the same process.

This behaviour can be modified and multiple applications grouped together using the WSGIApplicationGroup directive. Specifically, the directive indicates that the marked WSGI applications should be run within the context of a common sub interpreter rather than being run in their own sub interpreters. By doing this, each WSGI application will then have access to the same global data. Do note though that this doesn't change the fact that global data will not be shared between processes.

The only other way of sharing data between sub interpreters within the one child process would be to use an external data store, or a third party C extension module for Python which allows communication or sharing of data between multiple interpreters within the same process.

Building A Portable Application

Taking into consideration the different process models used by Apache and the manner in which interpreters are used by mod_wsgi, to build a portable and robust application requires the following therefore be satisified.

1. Where shared data needs to be visible to all application instances, regardless of which child process they execute in, and changes made to the data by one application are immediately available to another, including any executing in another child process, an external data store such as a database or shared memory must be used. Global variables in normal Python modules cannot be used for this purpose.

2. Access to and modification of shared data in an external data store must be protected so as to prevent multiple threads in the same or different processes from interfering with each other. This would normally be achieved through a locking mechanism visible to all child processes.

3. An application must be re-entrant, or simply put, be able to be called concurrently by multiple threads at the same time. Data which needs to exist for the life of the request, would need to be stored as stack based data, thread local data, or cached in the WSGI application environment. Global variables within the actual application module cannot be used for this purpose.

4. Where global data in a module local to a child process is still used, for example as a cache, access to and modification of the global data must be protected by local thread locking mechanisms.