Apache Performance Tuning easiest way

Apache 2.x is a general-purpose webserver, designed to provide a balance of flexibility, portability, and performance. Although it has not been designed specifically to set benchmark records, Apache 2.x is capable of high performance in many real-world situations.

Compared to Apache 1.3, release 2.x contains many additional optimizations to increase throughput and scalability. Most of these improvements are enabled by default. However, there are compile-time and run-time configuration choices that can significantly affect performance. This document describes the options that a server administrator can configure to tune the performance of an Apache 2.x installation. Some of these configuration options enable the httpd to better take advantage of the capabilities of the hardware and OS, while others allow the administrator to trade functionality for speed.

For Extra Things , READ APACHE GUIDE, from

apache.org

Hardware and Operating System Issues

The single biggest hardware issue affecting webserver performance is RAM. A webserver should never ever have to swap, as swapping increases the latency of each request beyond a point that users consider “fast enough”. This causes users to hit stop and reload, further increasing the load. You can, and should, control the MaxClients setting so that your server does not spawn so many children it starts swapping. This procedure for doing this is simple: determine the size of your average Apache process, by looking at your process list via a tool such as top, and divide this into your total available memory, leaving some room for other processes.

Beyond that the rest is mundane: get a fast enough CPU, a fast enough network card, and fast enough disks, where “fast enough” is something that needs to be determined by experimentation.

Operating system choice is largely a matter of local concerns. But some guidelines that have proven generally useful are:

- Run the latest stable release and patchlevel of the operating system that you choose. Many OS suppliers have introduced significant performance improvements to their TCP stacks and thread libraries in recent years.
- If your OS supports a sendfile(2) system call, make sure you install the release and/or patches needed to enable it. (With Linux, for example, this means using Linux 2.4 or later. For early releases of Solaris 8, you may need to apply a patch.) On systems where it is available, sendfile enables Apache 2 to deliver static content faster and with lower CPU utilization.

Run-Time Configuration Issues

Related Modules	Related Directives

• mod_dir • mpm_common • mod_status	• AllowOverride • DirectoryIndex • HostnameLookups • EnableMMAP • EnableSendfile • KeepAliveTimeout • MaxSpareServers • MinSpareServers • Options • StartServers

HostnameLookups and other DNS considerations

Prior to Apache 1.3, HostnameLookups defaulted to On. This adds latency to every request because it requires a DNS lookup to complete before the request is finished. In Apache 1.3 this setting defaults to Off. If you need to have addresses in your log files resolved to hostnames, use the logresolveprogram that comes with Apache, or one of the numerous log reporting packages which are available.

It is recommended that you do this sort of postprocessing of your log files on some machine other than the production web server machine, in order that this activity not adversely affect server performance.

If you use any Allow from domain or Deny from domain directives (i.e., using a hostname, or a domain name, rather than an IP address) then you will pay for two DNS lookups (a reverse, followed by a forward lookup to make sure that the reverse is not being spoofed). For best performance, therefore, use IP addresses, rather than names, when using these directives, if possible.

Note that it's possible to scope the directives, such as within a section. In this case the DNS lookups are only performed on requests matching the criteria. Here’s an example which disables lookups except for .html and .cgi files:

```
HostnameLookups off

HostnameLookups on
```

But even still, if you just need DNS names in some CGIs you could consider doing the gethostbyname call in the specific CGIs that need it.

FollowSymLinks and SymLinksIfOwnerMatch

Wherever in your URL-space you do not have an Options FollowSymLinks, or you do have an Options SymLinksIfOwnerMatch Apache will have to issue extra system calls to check up on symlinks. One extra call per filename component. For example, if you had:

```
DocumentRoot /www/htdocs

Options SymLinksIfOwnerMatch
```

and a request is made for the URI /index.html. Then Apache will perform lstat(2) on /www, /www/htdocs, and /www/htdocs/index.html. The results of these lstats are never cached, so they will occur on every single request. If you really desire the symlinks security checking you can do something like this:

```
DocumentRoot /www/htdocs

Options FollowSymLinks


Options -FollowSymLinks +SymLinksIfOwnerMatch
```

This at least avoids the extra checks for the DocumentRoot path. Note that you’ll need to add similar sections if you have any Alias or RewriteRule paths outside of your document root. For highest performance, and no symlink protection, set FollowSymLinks everywhere, and never setSymLinksIfOwnerMatch.

AllowOverride

Wherever in your URL-space you allow overrides (typically .htaccess files) Apache will attempt to open .htaccess for each filename component. For example,

```
DocumentRoot /www/htdocs

AllowOverride all
```

and a request is made for the URI /index.html. Then Apache will attempt to open /.htaccess, /www/.htaccess, and /www/htdocs/.htaccess. The solutions are similar to the previous case of Options FollowSymLinks. For highest performance use AllowOverride None everywhere in your filesystem.

Negotiation

If at all possible, avoid content-negotiation if you’re really interested in every last ounce of performance. In practice the benefits of negotiation outweigh the performance penalties. There’s one case where you can speed up the server. Instead of using a wildcard such as:

```
DirectoryIndex index
```

Use a complete list of options:

```
DirectoryIndex index.cgi index.pl index.shtml index.html
```

where you list the most common choice first.

Also note that explicitly creating a type-map file provides better performance than using MultiViews, as the necessary information can be determined by reading this single file, rather than having to scan the directory for files.

If your site needs content negotiation consider using type-map files, rather than the Options MultiViews directive to accomplish the negotiation. See theContent Negotiation documentation for a full discussion of the methods of negotiation, and instructions for creating type-map files.

Memory-mapping

In situations where Apache 2.x needs to look at the contents of a file being delivered—for example, when doing server-side-include processing—it normally memory-maps the file if the OS supports some form of mmap(2).

On some platforms, this memory-mapping improves performance. However, there are cases where memory-mapping can hurt the performance or even the stability of the httpd:

- On some operating systems, mmap does not scale as well as read(2) when the number of CPUs increases. On multiprocessor Solaris servers, for example, Apache 2.x sometimes delivers server-parsed files faster when mmap is disabled.
- If you memory-map a file located on an NFS-mounted filesystem and a process on another NFS client machine deletes or truncates the file, your process may get a bus error the next time it tries to access the mapped file content.

For installations where either of these factors applies, you should use EnableMMAP off to disable the memory-mapping of delivered files. (Note: This directive can be overridden on a per-directory basis.)

Sendfile

In situations where Apache 2.x can ignore the contents of the file to be delivered – for example, when serving static file content – it normally uses the kernel sendfile support the file if the OS supports the sendfile(2) operation.

On most platforms, using sendfile improves performance by eliminating separate read and send mechanics. However, there are cases where using sendfile can harm the stability of the httpd:

- Some platforms may have broken sendfile support that the build system did not detect, especially if the binaries were built on another box and moved to such a machine with broken sendfile support.
- With an NFS-mounted files, the kernel may be unable to reliably serve the network file through it's own cache.

For installations where either of these factors applies, you should use EnableSendfile off to disable sendfile delivery of file contents. (Note: This directive can be overridden on a per-directory basis.)

Process Creation

Prior to Apache 1.3 the MinSpareServers, MaxSpareServers, and StartServers settings all had drastic effects on benchmark results. In particular, Apache required a "ramp-up" period in order to reach a number of children sufficient to serve the load being applied. After the initial spawning ofStartServers children, only one child per second would be created to satisfy the MinSpareServers setting. So a server being accessed by 100 simultaneous clients, using the default StartServers of 5 would take on the order 95 seconds to spawn enough children to handle the load. This works fine in practice on real-life servers, because they aren't restarted frequently. But does really poorly on benchmarks which might only run for ten minutes.

The one-per-second rule was implemented in an effort to avoid swamping the machine with the startup of new children. If the machine is busy spawning children it can’t service requests. But it has such a drastic effect on the perceived performance of Apache that it had to be replaced. As of Apache 1.3, the code will relax the one-per-second rule. It will spawn one, wait a second, then spawn two, wait a second, then spawn four, and it will continue exponentially until it is spawning 32 children per second. It will stop whenever it satisfies the MinSpareServers setting.

This appears to be responsive enough that it’s almost unnecessary to twiddle the MinSpareServers, MaxSpareServers and StartServers knobs. When more than 4 children are spawned per second, a message will be emitted to the ErrorLog. If you see a lot of these errors then consider tuning these settings. Use the mod_status output as a guide.

Related to process creation is process death induced by the MaxRequestsPerChild setting. By default this is 0, which means that there is no limit to the number of requests handled per child. If your configuration currently has this set to some very low number, such as 30, you may want to bump this up significantly. If you are running SunOS or an old version of Solaris, limit this to 10000 or so because of memory leaks.

When keep-alives are in use, children will be kept busy doing nothing waiting for more requests on the already open connection. The defaultKeepAliveTimeout of 5 seconds attempts to minimize this effect. The tradeoff here is between network bandwidth and server resources. In no event should you raise this above about 60 seconds, as most of the benefits are lost.

Compile-Time Configuration Issues

Choosing an MPM

Apache 2.x supports pluggable concurrency models, called Multi-Processing Modules (MPMs). When building Apache, you must choose an MPM to use. There are platform-specific MPMs for some platforms: beos, mpm_netware, mpmt_os2, and mpm_winnt. For general Unix-type systems, there are several MPMs from which to choose. The choice of MPM can affect the speed and scalability of the httpd:

- The worker MPM uses multiple child processes with many threads each. Each thread handles one connection at a time. Worker generally is a good choice for high-traffic servers because it has a smaller memory footprint than the prefork MPM.
- The prefork MPM uses multiple child processes with one thread each. Each process handles one connection at a time. On many systems, prefork is comparable in speed to worker, but it uses more memory. Prefork's threadless design has advantages over worker in some situations: it can be used with non-thread-safe third-party modules, and it is easier to debug on platforms with poor thread debugging support.

For more information on these and other MPMs, please see the MPM documentation.

Modules

Since memory usage is such an important consideration in performance, you should attempt to eliminate modules that you are not actually using. If you have built the modules as DSOs, eliminating modules is a simple matter of commenting out the associated LoadModule directive for that module. This allows you to experiment with removing modules, and seeing if your site still functions in their absense.

If, on the other hand, you have modules statically linked into your Apache binary, you will need to recompile Apache in order to remove unwanted modules.

An associated question that arises here is, of course, what modules you need, and which ones you don’t. The answer here will, of course, vary from one web site to another. However, the *minimal* list of modules which you can get by with tends to include mod_mime, mod_dir, and mod_log_config.mod_log_config is, of course, optional, as you can run a web site without log files. This is, however, not recommended.

Atomic Operations

Some modules, such as mod_cache and recent development builds of the worker MPM, use APR’s atomic API. This API provides atomic operations that can be used for lightweight thread synchronization.

By default, APR implements these operations using the most efficient mechanism available on each target OS/CPU platform. Many modern CPUs, for example, have an instruction that does an atomic compare-and-swap (CAS) operation in hardware. On some platforms, however, APR defaults to a slower, mutex-based implementation of the atomic API in order to ensure compatibility with older CPU models that lack such instructions. If you are building Apache for one of these platforms, and you plan to run only on newer CPUs, you can select a faster atomic implementation at build time by configuring Apache with the --enable-nonportable-atomics option:

```
./buildconf
./configure --with-mpm=worker --enable-nonportable-atomics=yes
```

The --enable-nonportable-atomics option is relevant for the following platforms:

- Solaris on SPARC
  By default, APR uses mutex-based atomics on Solaris/SPARC. If you configure with --enable-nonportable-atomics, however, APR generates code that uses a SPARC v8plus opcode for fast hardware compare-and-swap. If you configure Apache with this option, the atomic operations will be more efficient (allowing for lower CPU utilization and higher concurrency), but the resulting executable will run only on UltraSPARC chips.
- Linux on x86
  By default, APR uses mutex-based atomics on Linux. If you configure with --enable-nonportable-atomics, however, APR generates code that uses a 486 opcode for fast hardware compare-and-swap. This will result in more efficient atomic operations, but the resulting executable will run only on 486 and later chips (and not on 386).

mod_status and ExtendedStatus On

If you include mod_status and you also set ExtendedStatus On when building and running Apache, then on every request Apache will perform two calls to gettimeofday(2) (or times(2) depending on your operating system), and (pre-1.3) several extra calls to time(2). This is all done so that the status report contains timing indications. For highest performance, set ExtendedStatus off (which is the default).

accept Serialization – multiple sockets

Warning:

This section has not been fully updated to take into account changes made in the 2.x version of the Apache HTTP Server.

Some of the information may still be relevant, but please use it with care.

This discusses a shortcoming in the Unix socket API. Suppose your web server uses multiple Listen statements to listen on either multiple ports or multiple addresses. In order to test each socket to see if a connection is ready Apache uses select(2). select(2) indicates that a socket has *zero* or *at least one* connection waiting on it. Apache's model includes multiple children, and all the idle ones test for new connections at the same time. A naive implementation looks something like this (these examples do not match the code, they're contrived for pedagogical purposes):

```
for (;;) {
for (;;) {
fd_set accept_fds;

FD_ZERO (&accept_fds);
for (i = first_socket; i <= last_socket; ++i) { FD_SET (i,
&accept_fds); } rc = select (last_socket+1, &accept_fds, NULL,
NULL, NULL); if (rc < 1) continue; new_connection = -1; for (i
= first_socket; i <= last_socket; ++i) { if (FD_ISSET (i,
&accept_fds)) { new_connection = accept (i, NULL, NULL); if
(new_connection != -1) break; } } if (new_connection != -1)
break; } process the new_connection; }
```

But this naive implementation has a serious starvation problem. Recall that multiple children execute this loop at the same time, and so multiple children will block at select when they are in between requests. All those blocked children will awaken and return from select when a single request appears on any socket (the number of children which awaken varies depending on the operating system and timing issues). They will all then fall down into the loop and try to accept the connection. But only one will succeed (assuming there's still only one connection ready), the rest will be *blocked* in accept. This effectively locks those children into serving requests from that one socket and no other sockets, and they'll be stuck there until enough new requests appear on that socket to wake them all up. This starvation problem was first documented in PR#467. There are at least two solutions.

One solution is to make the sockets non-blocking. In this case the accept won't block the children, and they will be allowed to continue immediately. But this wastes CPU time. Suppose you have ten idle children in select, and one connection arrives. Then nine of those children will wake up, try to accept the connection, fail, and loop back into select, accomplishing nothing. Meanwhile none of those children are servicing requests that occurred on other sockets until they get back up to the select again. Overall this solution does not seem very fruitful unless you have as many idle CPUs (in a multiprocessor box) as you have idle children, not a very likely situation.

Another solution, the one used by Apache, is to serialize entry into the inner loop. The loop looks like this (differences highlighted):

```
for (;;) {
accept_mutex_on ();
for (;;) {
fd_set accept_fds;

FD_ZERO (&accept_fds);
for (i = first_socket; i <= last_socket; ++i) { FD_SET (i,
&accept_fds); } rc = select (last_socket+1, &accept_fds, NULL,
NULL, NULL); if (rc < 1) continue; new_connection = -1; for (i
= first_socket; i <= last_socket; ++i) { if (FD_ISSET (i,
&accept_fds)) { new_connection = accept (i, NULL, NULL); if
(new_connection != -1) break; } } if (new_connection != -1)
break; } accept_mutex_off ();
process the new_connection;
}
```

The functions accept_mutex_on and accept_mutex_off implement a mutual exclusion semaphore. Only one child can have the mutex at any time. There are several choices for implementing these mutexes. The choice is defined in src/conf.h (pre-1.3) or src/include/ap_config.h (1.3 or later). Some architectures do not have any locking choice made, on these architectures it is unsafe to use multiple Listen directives.

The directive AcceptMutex can be used to change the selected mutex implementation at run-time.

AcceptMutex flock
: This method uses the flock(2) system call to lock a lock file (located by the LockFile directive).

AcceptMutex fcntl
: This method uses the fcntl(2) system call to lock a lock file (located by the LockFile directive).

AcceptMutex sysvsem
: (1.3 or later) This method uses SysV-style semaphores to implement the mutex. Unfortunately SysV-style semaphores have some bad side-effects. One is that it’s possible Apache will die without cleaning up the semaphore (see the ipcs(8) man page). The other is that the semaphore API allows for a denial of service attack by any CGIs running under the same uid as the webserver (*i.e.*, all CGIs, unless you use something like suexec orcgiwrapper). For these reasons this method is not used on any architecture except IRIX (where the previous two are prohibitively expensive on most IRIX boxes).

AcceptMutex pthread
: (1.3 or later) This method uses POSIX mutexes and should work on any architecture implementing the full POSIX threads specification, however appears to only work on Solaris (2.5 or later), and even then only in certain configurations. If you experiment with this you should watch out for your server hanging and not responding. Static content only servers may work just fine.

AcceptMutex posixsem
: (2.0 or later) This method uses POSIX semaphores. The semaphore ownership is not recovered if a thread in the

process holding the mutex segfaults, resulting in a hang of the web server.

If your system has another method of serialization which isn’t in the above list then it may be worthwhile adding code for it to APR.

Another solution that has been considered but never implemented is to partially serialize the loop — that is, let in a certain number of processes. This would only be of interest on multiprocessor boxes where it’s possible multiple children could run simultaneously, and the serialization actually doesn’t take advantage of the full bandwidth. This is a possible area of future investigation, but priority remains low because highly parallel web servers are not the norm.

Ideally you should run servers without multiple Listen statements if you want the highest performance. But read on.

accept Serialization – single socket

The above is fine and dandy for multiple socket servers, but what about single socket servers? In theory they shouldn’t experience any of these same problems because all children can just block in accept(2) until a connection arrives, and no starvation results. In practice this hides almost the same “spinning” behaviour discussed above in the non-blocking solution. The way that most TCP stacks are implemented, the kernel actually wakes up all processes blocked in accept when a single connection arrives. One of those processes gets the connection and returns to user-space, the rest spin in the kernel and go back to sleep when they discover there’s no connection for them. This spinning is hidden from the user-land code, but it’s there nonetheless. This can result in the same load-spiking wasteful behaviour that a non-blocking solution to the multiple sockets case can.

For this reason we have found that many architectures behave more “nicely” if we serialize even the single socket case. So this is actually the default in almost all cases. Crude experiments under Linux (2.0.30 on a dual Pentium pro 166 w/128Mb RAM) have shown that the serialization of the single socket case causes less than a 3% decrease in requests per second over unserialized single-socket. But unserialized single-socket showed an extra 100ms latency on each request. This latency is probably a wash on long haul lines, and only an issue on LANs. If you want to override the single socket serialization you can define SINGLE_LISTEN_UNSERIALIZED_ACCEPT and then single-socket servers will not serialize at all.

Lingering Close

As discussed in draft-ietf-http-connection-00.txt section 8, in order for an HTTP server to **reliably** implement the protocol it needs to shutdown each direction of the communication independently (recall that a TCP connection is bi-directional, each half is independent of the other). This fact is often overlooked by other servers, but is correctly implemented in Apache as of 1.2.

When this feature was added to Apache it caused a flurry of problems on various versions of Unix because of a shortsightedness. The TCP specification does not state that the FIN_WAIT_2 state has a timeout, but it doesn’t prohibit it. On systems without the timeout, Apache 1.2 induces many sockets stuck forever in the FIN_WAIT_2 state. In many cases this can be avoided by simply upgrading to the latest TCP/IP patches supplied by the vendor. In cases where the vendor has never released patches (*i.e.*, SunOS4 – although folks with a source license can patch it themselves) we have decided to disable this feature.

There are two ways of accomplishing this. One is the socket option SO_LINGER. But as fate would have it, this has never been implemented properly in most TCP/IP stacks. Even on those stacks with a proper implementation (*i.e.*, Linux 2.0.31) this method proves to be more expensive (cputime) than the next solution.

For the most part, Apache implements this in a function called lingering_close (in http_main.c). The function looks roughly like this:

```
void lingering_close (int s)
{
char junk_buffer[2048];

/* shutdown the sending side */
shutdown (s, 1);

signal (SIGALRM, lingering_death);
alarm (30);

for (;;) {
select (s for reading, 2 second timeout);
if (error) break;
if (s is ready for reading) {
if (read (s, junk_buffer, sizeof (junk_buffer)) <= 0) { break;
} /* just toss away whatever is here */ } } close (s); }
```

This naturally adds some expense at the end of a connection, but it is required for a reliable implementation. As HTTP/1.1 becomes more prevalent, and all connections are persistent, this expense will be amortized over more requests. If you want to play with fire and disable this feature you can defineNO_LINGCLOSE, but this is not recommended at all. In particular, as HTTP/1.1 pipelined persistent connections come into use lingering_close is an absolute necessity (and pipelined connections are faster, so you want to support them).

Scoreboard File

Apache’s parent and children communicate with each other through something called the scoreboard. Ideally this should be implemented in shared memory. For those operating systems that we either have access to, or have been given detailed ports for, it typically is implemented using shared memory. The rest default to using an on-disk file. The on-disk file is not only slow, but it is unreliable (and less featured). Peruse the src/main/conf.h file for your architecture and look for either USE_MMAP_SCOREBOARD or USE_SHMGET_SCOREBOARD. Defining one of those two (as well as their companions HAVE_MMAPand HAVE_SHMGET respectively) enables the supplied shared memory code. If your system has another type of shared memory, edit the filesrc/main/http_main.c and add the hooks necessary to use it in Apache. (Send us back a patch too please.)

Historical note: The Linux port of Apache didn’t start to use shared memory until version 1.2 of Apache. This oversight resulted in really poor and unreliable behaviour of earlier versions of Apache on Linux.

DYNAMIC_MODULE_LIMIT

If you have no intention of using dynamically loaded modules (you probably don’t if you’re reading this and tuning your server for every last ounce of performance) then you should add -DDYNAMIC_MODULE_LIMIT=0 when building your server. This will save RAM that’s allocated only for supporting dynamically loaded modules.

Appendix: Detailed Analysis of a

Trace

Here is a system call trace of Apache 2.0.38 with the worker MPM on Solaris 8. This trace was collected using:

truss -l -p *httpd_child_pid*.

The -l option tells truss to log the ID of the LWP (lightweight process–Solaris's form of kernel-level thread) that invokes each system call.

Other systems may have different system call tracing utilities such as strace, ktrace, or par. They all produce similar output.

In this trace, a client has requested a 10KB static file from the httpd. Traces of non-static requests or requests with content negotiation look wildly different (and quite ugly in some cases).

[crayon-669e1d74c969d219548859/]
In this trace, the listener thread is running within LWP #67.

Note the lack of accept(2) serialization. On this particular platform, the worker MPM uses an unserialized accept by default unless it is listening on multiple ports.
[crayon-669e1d74c96a8164098607/]
Upon accepting the connection, the listener thread wakes up a worker thread to do the request processing. In this trace, the worker thread that handles the request is mapped to LWP #65.

[crayon-669e1d74c96aa761226679/]
In order to implement virtual hosts, Apache needs to know the local socket address used to accept the connection. It is possible to eliminate this call in many situations (such as when there are no virtual hosts, or when Listen directives are used which do not have wildcard addresses). But no effort has yet been made to do these optimizations.

[crayon-669e1d74c96ad726799720/]

The brk(2) calls allocate memory from the heap. It is rare to see these in a system call trace, because the httpd uses custom memory allocators (apr_pool and apr_bucket_alloc) for most request processing. In this trace, the httpd has just been started, so it must call malloc(3) to get the blocks of raw memory with which to create the custom memory allocators.

[crayon-669e1d74c96af496788605/]
Next, the worker thread puts the connection to the client (file descriptor 9) in non-blocking mode. The setsockopt(2) and getsockopt(2) calls are a side-effect of how Solaris's libc handles fcntl(2) on sockets.

[crayon-669e1d74c96b2754529974/]
The worker thread reads the request from the client.

[crayon-669e1d74c96b4324520746/]
This httpd has been configured with Options FollowSymLinks and AllowOverride None. Thus it doesn't need to lstat(2) each directory in the path leading up to the requested file, nor check for .htaccess files. It simply calls stat(2) to verify that the file: 1) exists, and 2) is a regular file, not a directory.

[crayon-669e1d74c96b7456843874/]
In this example, the httpd is able to send the HTTP response header and the requested file with a single sendfilev(2) system call. Sendfile semantics vary among operating systems. On some other systems, it is necessary to do a write(2) or writev(2) call to send the headers before callingsendfile(2).

[crayon-669e1d74c96b9979617953/]
This write(2) call records the request in the access log. Note that one thing missing from this trace is a time(2) call. Unlike Apache 1.3, Apache 2.x usesgettimeofday(3) to look up the time. On some operating systems, like Linux or Solaris, gettimeofday has an optimized implementation that doesn't require as much overhead as a typical system call.

[crayon-669e1d74c96bc090635904/]
The worker thread does a lingering close of the connection.

[crayon-669e1d74c96be797933367/]
Finally the worker thread closes the file that it has just delivered and blocks until the listener assigns it another connection.

[crayon-669e1d74c96c1808870935/]
Meanwhile, the listener thread is able to accept another connection as soon as it has dispatched this connection to a worker thread (subject to some flow-control logic in the worker MPM that throttles the listener if all the available workers are busy). Though it isn’t apparent from this trace, the next accept(2)can (and usually does, under high load conditions) occur in parallel with the worker thread’s handling of the just-accepted connection.